Iptables-Semantics

Cornelius Diekmann, Lars Hupel

February 6, 2026

Abstract

We present a big step semantics of the filtering behavior of the Linux/netfilter iptables firewall. We provide algorithms to simplify complex iptables rulests to a simple firewall model (c.f. AFP entry Simple_Firewall) and to verify spoofing protection of a ruleset. Internally, we embed our semantics into ternary logic, ultimately supporting every iptables match condition by abstracting over unknowns. Using this AFP entry and all entries it depends on, we created an easy-to-use, stand-alone haskell tool called *fffuu* (http://iptables.isabelle.systems). The tool does not require any input —except for the `iptables-save` dump of the analyzed firewall— and presents interesting results about the user's ruleset. Real-Word firewall errors have been uncovered, as well as the correctness of rulesets has been proven with the help of our tool.

For a detailed description, see [2, 4, 3, 1].

Acknowledgements This entry would not have been possible without the help of Julius Michaelis, Max Haslbeck, Stephan-A. Posselt, Lars Noschinski, Manuel Eberl, Gerwin Klein, the Isabelle group Munich, and Georg Carle.

Contents

1 Repeat finitely Until it Stabilizes **5**

2 Firewall Basic Syntax **6**

3 Basic Algorithms **7**

4 Big Step Semantics **10**
4.1 Boolean Matcher Algebra 37
4.2 Add match 40
4.3 Background Ruleset Updating 45

5 *Call Return* Unfolding **54**
5.1 Completeness . . . 56
5.2 *process-ret* correctness . . . 62
5.3 Soundness . . . 69

6 Ternary Logic **73**
6.1 Negation Normal Form . . . 78

7 Packet Matching in Ternary Logic **79**
7.1 Ternary Matcher Algebra . . . 82
7.2 Removing Unknown Primitives . . . 84

8 Embedded Ternary-Matching Big Step Semantics **90**
8.1 Ternary Semantics (Big Step) . . . 90
8.2 wf ruleset . . . 94
8.3 Ternary Semantics (Function) . . . 94
8.3.1 Append, Prepend, Postpend, Composition . . . 96
8.4 Equality with γ,$p\vdash$ $\langle rs,\ s\rangle \Rightarrow_\alpha t$ semantics . . . 98
8.5 Matching . . . 102

9 IPv4 Addresses **105**
9.1 IPv4 Addresses in IPTables Notation (how we parse it) . . . 105

10 Matching TCP Flags **109**

11 Ports (layer 4) **113**

12 Tagged Simple Packet **115**

13 Primitive Matchers: Interfaces, IP Space, Layer 4 Ports Matcher **117**

14 Approximate Matching Tactics **118**
14.1 A Generic primitive matcher: Agnostic of IP Addresses . . . 119
14.2 Basic optimisations . . . 122
14.3 Primitive Matchers: IP Port Iface Matcher . . . 123
14.4 Basic optimisations . . . 126
14.5 Abstracting over unknowns . . . 127

15 Examples Big Step Semantics **129**

16 Semantics Stateful **144**
16.1 Model 1 – Curried Stateful Matcher . . . 144
16.2 Model 2 – Packets Tagged with State Information . . . 145
16.3 Example: Conntrack with curried matcher . . . 148
16.4 Example: Conntrack with packet tagging . . . 151

17 Big Step Semantics with Goto **152**
17.1 Semantics . . . 152
17.1.1 Forward reasoning . . . 156
17.2 Determinism . . . 165
17.3 Matching . . . 166
17.4 Goto Unfolding . . . 172

18 Negation Type DNF **180**
18.0.1 inverting a DNF . . . 182
18.0.2 Optimizing . . . 183

19 Boolean Matching vs. Ternary Matching **183**

20 Fixed Action **187**
20.1 *match-list* . . . 192

21 Normalized (DNF) matches **196**

22 Normalizing rules instead of only match expressions **199**
22.1 Functions which preserve *normalized-nnf-match* . . . 204

23 Negation Type Matching **205**

24 Primitive Normalization **208**
24.1 Normalized Primitives . . . 208
24.2 Primitive Extractor . . . 213
24.3 Normalizing and Optimizing Primitives . . . 219
24.4 Optimizing a match expression . . . 225
24.5 Processing a list of normalization functions . . . 230

25 Combine Match Expressions **232**

26 Further Lemmas about the Common Matcher **234**

27 Normalizing L4 Ports **234**
27.1 Defining Normalized Ports . . . 234
27.2 Compressing Positive Matches on Ports into a Single Match . 235
27.3 Rewriting Negated Matches on Ports . . . 237
27.4 Normalizing Positive Matches on Ports . . . 245
27.5 Complete Normalization . . . 250
27.6 Normalizing IP Addresses . . . 265
27.7 Optimizing interfaces in match expressions . . . 268

28 Word Upto **273**

29 Optimizing Protocols **278**

30 Optimizing protocols in match expressions **278**
30.1 Importing the matches on *primitive-protocol* from *L4Ports* . . 283
30.2 Putting things together 288

31 Reverse Remdups **290**
31.1 Sanity checking for an *'i ipassignment*. 293
31.2 IP Assignment difference 301

32 No Spoofing **302**
32.1 Spoofing Protection 302

33 Firewall toString Functions **322**

34 Routing and IP Assignments **325**
34.1 Routing IP Assignment 325

35 Replacing output interfaces by their IP ranges according to Routing **326**

36 Trying to connect inbound interfaces by their IP ranges **331**
36.1 Constraining Interfaces 331
36.2 Sanity checking the assumption 335
36.3 Replacing Interfaces Completely 336

37 Optimizing **340**
37.1 Removing Shadowed Rules 340
37.1.1 Soundness 340
37.2 Removing rules which cannot apply 342

38 Optimizing and Normalizing Primitives **343**

39 Transforming rulesets **347**
39.1 Optimizations 347
39.2 Optimize and Normalize to NNF form 348
39.3 Abstracting over unknowns 352
39.4 Normalizing and Transforming Primitives 356

40 Abstracting over Primitives **391**

41 Iptables to Simple Firewall and Vice Versa **401**
41.1 Simple Match to MatchExpr 401
41.2 MatchExpr to Simple Match 402
41.2.1 Normalizing Interfaces 404
41.2.2 Normalizing Protocols 404

42 Semantics Embedding **424**
42.1 Tactic *in-doubt-allow* . . . 424
42.2 Tactic *in-doubt-deny* . . . 427
42.3 Approximating Closures . . . 430
42.4 Exact Embedding . . . 430

43 Normalizing Rulesets in the Boolean Big Step Semantics **432**

44 Code Interface **433**
44.1 L4 Ports Parser Helper . . . 436

45 Parser for iptables-save **437**

46 An SML Parser for iptables-save **438**

47 Spoofing protection in Ternary Semantics implies Spoofing protection Boolean Semantics **452**

48 Parser for iptables-save **455**

49 An SML Parser for iptables-save **455**

50 Applying the Access Matrix to the Bigstep Semantics **471**

51 Documentation **473**
51.1 General Model . . . 473
51.2 Unfolding the Ruleset . . . 474
51.3 Spoofing protection . . . 475
51.4 Simple Firewall Model . . . 476
51.5 Service Matrices . . . 477

1 Repeat finitely Until it Stabilizes

theory *Repeat-Stabilize*
imports *Main*
begin

Repeating something a number of times

Iterating a function at most n times (first parameter) until it stabilizes.

fun *repeat-stabilize* :: *nat* ⇒ ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a* **where**
repeat-stabilize 0 - v = v |
repeat-stabilize (*Suc n*) *f v* = (*let v-new* = *f v in if v* = *v-new then v else repeat-stabilize n f v-new*)

lemma *repeat-stabilize-funpow*: *repeat-stabilize n f v* = (*f* ^^ *n*) *v*
proof(*induction n arbitrary*: *v*)

case (*Suc n*)
have $f\ v = v \Longrightarrow (f\widehat{\ }n)\ v = v$ **by**(*induction n*) *simp-all*
with *Suc* **show** *?case* **by**(*simp add*: *Let-def funpow-swap1*)
qed(*simp*)

lemma *repeat-stabilize-induct*: $(P\ m) \Longrightarrow (\bigwedge m.\ P\ m \Longrightarrow P\ (f\ m)) \Longrightarrow P$ (*repeat-stabilize n f m*)
apply(*simp add*: *repeat-stabilize-funpow*)
apply(*induction n*)
by(*simp*)+

end

2 Firewall Basic Syntax

theory *Firewall-Common*
imports *Main Simple-Firewall.Firewall-Common-Decision-State Common/Repeat-Stabilize*
begin

Our firewall model supports the following actions.

datatype *action* = *Accept* | *Drop* | *Log* | *Reject* | *Call string* | *Return* | *Goto string* | *Empty* | *Unknown*

We support the following algebra over primitives of type *'a*. The type parameter *'a* denotes the primitive match condition. For example, matching on source IP address or on protocol. We lift the primitives to an algebra. Note that we do not have an Or expression.

datatype *'a match-expr* = *Match 'a*
| *MatchNot 'a match-expr*
| *MatchAnd 'a match-expr 'a match-expr*
| *MatchAny*

definition *MatchOr* :: *'a match-expr* ⇒ *'a match-expr* ⇒ *'a match-expr* **where**
MatchOr m1 m2 = *MatchNot* (*MatchAnd* (*MatchNot m1*) (*MatchNot m2*))

A firewall rule consists of a match expression and an action.

datatype *'a rule* = *Rule* (*get-match*: *'a match-expr*) (*get-action*: *action*)

lemma *rules-singleton-rev-E*:
$[Rule\ m\ a] = rs_1\ @\ rs_2 \Longrightarrow$
$(rs_1 = [Rule\ m\ a] \Longrightarrow rs_2 = [] \Longrightarrow P\ m\ a) \Longrightarrow$
$(rs_1 = [] \Longrightarrow rs_2 = [Rule\ m\ a] \Longrightarrow P\ m\ a) \Longrightarrow P\ m\ a$
by (*cases* rs_1) *auto*

3 Basic Algorithms

These algorithms should be valid for all firewall semantics. The corresponding proofs follow once the semantics are defined.

The actions Log and Empty do not modify the packet processing in any way. They can be removed.

fun *rm-LogEmpty* :: *'a rule list* ⇒ *'a rule list* **where**
rm-LogEmpty [] = [] |
rm-LogEmpty ((*Rule - Empty*)#*rs*) = *rm-LogEmpty rs* |
rm-LogEmpty ((*Rule - Log*)#*rs*) = *rm-LogEmpty rs* |
rm-LogEmpty (*r*#*rs*) = *r* # *rm-LogEmpty rs*

lemma *rm-LogEmpty-filter*: *rm-LogEmpty rs* = *filter* (λ*r*. *get-action r* ≠ *Log* ∧ *get-action r* ≠ *Empty*) *rs*
by(*induction rs rule*: *rm-LogEmpty.induct*) (*simp-all*)

lemma *rm-LogEmpty-seq*: *rm-LogEmpty* (*rs1*@*rs2*) = *rm-LogEmpty rs1* @ *rm-LogEmpty rs2*
by(*simp add*: *rm-LogEmpty-filter*)

Optimize away MatchAny matches

fun *opt-MatchAny-match-expr-once* :: *'a match-expr* ⇒ *'a match-expr* **where**
opt-MatchAny-match-expr-once MatchAny = *MatchAny* |
opt-MatchAny-match-expr-once (*Match a*) = (*Match a*) |
opt-MatchAny-match-expr-once (*MatchNot* (*MatchNot m*)) = (*opt-MatchAny-match-expr-once m*) |
opt-MatchAny-match-expr-once (*MatchNot m*) = *MatchNot* (*opt-MatchAny-match-expr-once m*) |
opt-MatchAny-match-expr-once (*MatchAnd MatchAny MatchAny*) = *MatchAny* |
opt-MatchAny-match-expr-once (*MatchAnd MatchAny m*) = (*opt-MatchAny-match-expr-once m*) |

opt-MatchAny-match-expr-once (*MatchAnd m MatchAny*) = (*opt-MatchAny-match-expr-once m*) |
opt-MatchAny-match-expr-once (*MatchAnd* - (*MatchNot MatchAny*)) = (*MatchNot MatchAny*) |
opt-MatchAny-match-expr-once (*MatchAnd* (*MatchNot MatchAny*) -) = (*MatchNot MatchAny*) |
opt-MatchAny-match-expr-once (*MatchAnd m1 m2*) = *MatchAnd* (*opt-MatchAny-match-expr-once m1*) (*opt-MatchAny-match-expr-once m2*)

It is still a good idea to apply *opt-MatchAny-match-expr-once* multiple times. Example:

lemma *MatchNot* (*opt-MatchAny-match-expr-once* (*MatchAnd MatchAny* (*MatchNot MatchAny*))) = *MatchNot* (*MatchNot MatchAny*) **by** *simp*
lemma *m* = (*MatchAnd* (*MatchAnd MatchAny MatchAny*) (*MatchAnd MatchAny MatchAny*)) ⟹

(*opt-MatchAny-match-expr-once*^^*2*) *m* ≠ *opt-MatchAny-match-expr-once m* **by**(*simp add*: *funpow-def*)

definition *opt-MatchAny-match-expr* :: *'a match-expr* ⇒ *'a match-expr* **where**
opt-MatchAny-match-expr m ≡ *repeat-stabilize 2 opt-MatchAny-match-expr-once m*

Rewrite *Reject* actions to *Drop* actions. If we just care about the filtering decision (*FinalAllow* or *FinalDeny*), they should be equal.

fun *rw-Reject* :: *'a rule list* ⇒ *'a rule list* **where**
rw-Reject [] = [] |
rw-Reject ((*Rule m Reject*)#*rs*) = (*Rule m Drop*)#*rw-Reject rs* |
rw-Reject (*r*#*rs*) = *r* # *rw-Reject rs*

We call a ruleset simple iff the only actions are *Accept* and *Drop*

definition *simple-ruleset* :: *'a rule list* ⇒ *bool* **where**
simple-ruleset rs ≡ ∀ *r* ∈ *set rs*. *get-action r* = *Accept* ∨ *get-action r* = *Drop*

lemma *simple-ruleset-tail*: *simple-ruleset* (*r*#*rs*) ⟹ *simple-ruleset rs* **by** (*simp add*: *simple-ruleset-def*)

lemma *simple-ruleset-append*: *simple-ruleset* (*rs*$_1$ @ *rs*$_2$) ⟷ *simple-ruleset rs*$_1$ ∧ *simple-ruleset rs*$_2$
by(*simp add*: *simple-ruleset-def*, *blast*)

Structural properties about match expressions

fun *has-primitive* :: *'a match-expr* ⇒ *bool* **where**
has-primitive MatchAny = *False* |
has-primitive (*Match a*) = *True* |
has-primitive (*MatchNot m*) = *has-primitive m* |
has-primitive (*MatchAnd m1 m2*) = (*has-primitive m1* ∨ *has-primitive m2*)

Is a match expression equal to the *MatchAny* expression? Only applicable if no primitives are in the expression.

fun *matcheq-matchAny* :: *'a match-expr* ⇒ *bool* **where**
matcheq-matchAny MatchAny ⟷ *True* |
matcheq-matchAny (*MatchNot m*) ⟷ ¬ (*matcheq-matchAny m*) |
matcheq-matchAny (*MatchAnd m1 m2*) ⟷ *matcheq-matchAny m1* ∧ *matcheq-matchAny m2* |
matcheq-matchAny (*Match* -) = *undefined*

fun *matcheq-matchNone* :: *'a match-expr* ⇒ *bool* **where**
matcheq-matchNone MatchAny = *False* |
matcheq-matchNone (*Match* -) = *False* |
matcheq-matchNone (*MatchNot MatchAny*) = *True* |
matcheq-matchNone (*MatchNot* (*Match* -)) = *False* |
matcheq-matchNone (*MatchNot* (*MatchNot m*)) = *matcheq-matchNone m* |
matcheq-matchNone (*MatchNot* (*MatchAnd m1 m2*)) ⟷ *matcheq-matchNone* (*MatchNot m1*) ∧ *matcheq-matchNone* (*MatchNot m2*) |

matcheq-matchNone (*MatchAnd m1 m2*) ⟷ *matcheq-matchNone m1* ∨ *matcheq-matchNone m2*

lemma *matachAny-matchNone*: ¬ *has-primitive m* ⟹ *matcheq-matchAny m* ⟷ ¬ *matcheq-matchNone m*
by(*induction m rule*: *matcheq-matchNone.induct*)(*simp-all*)

lemma *matcheq-matchNone-no-primitive*: ¬ *has-primitive m* ⟹ *matcheq-matchNone* (*MatchNot m*) ⟷ ¬ *matcheq-matchNone m*
by(*induction m rule*: *matcheq-matchNone.induct*) (*simp-all*)

optimizing match expressions

fun *optimize-matches-option* :: (′*a match-expr* ⇒ ′*a match-expr option*) ⇒ ′*a rule list* ⇒ ′*a rule list* **where**
optimize-matches-option - [] = [] |
optimize-matches-option f (*Rule m a*#*rs*) = (*case f m of None* ⇒ *optimize-matches-option f rs* | *Some m* ⇒ (*Rule m a*)#*optimize-matches-option f rs*)

lemma *optimize-matches-option-simple-ruleset*: *simple-ruleset rs* ⟹ *simple-ruleset* (*optimize-matches-option f rs*)
proof(*induction rs rule*:*optimize-matches-option.induct*)
qed(*simp-all add*: *simple-ruleset-def split*: *option.split*)

lemma *optimize-matches-option-preserves*:
(⋀ *r m*. *r* ∈ *set rs* ⟹ *f* (*get-match r*) = *Some m* ⟹ *P m*) ⟹
∀ *r* ∈ *set* (*optimize-matches-option f rs*). *P* (*get-match r*)
apply(*induction rs rule*: *optimize-matches-option.induct*)
apply(*simp*; *fail*)
apply(*simp split*: *option.split*)
by *fastforce*

lemma *optimize-matches-option-append*: *optimize-matches-option f* (*rs1*@*rs2*) = *optimize-matches-option f rs1* @ *optimize-matches-option f rs2*
proof(*induction rs1 rule*: *optimize-matches-option.induct*)
qed(*simp-all split*: *option.split*)

definition *optimize-matches* :: (′*a match-expr* ⇒ ′*a match-expr*) ⇒ ′*a rule list* ⇒ ′*a rule list* **where**
optimize-matches f rs = *optimize-matches-option* (λ*m*. (*if matcheq-matchNone* (*f m*) *then None else Some* (*f m*))) *rs*

lemma *optimize-matches-append*: *optimize-matches f* (*rs1*@*rs2*) = *optimize-matches f rs1* @ *optimize-matches f rs2*
by(*simp add*: *optimize-matches-def optimize-matches-option-append*)

lemma *optimize-matches-fst*: *optimize-matches f* (*r*#*rs*) = *optimize-matches f*

[r]@*optimize-matches f rs*
by(*cases r*)(*simp add*: *optimize-matches-def*)

lemma *optimize-matches-preserves*: (⋀ *r*. *r* ∈ *set rs* ⟹ *P* (*f* (*get-match r*))) ⟹
∀ *r* ∈ *set* (*optimize-matches f rs*). *P* (*get-match r*)
unfolding *optimize-matches-def*
apply(*rule optimize-matches-option-preserves*)
by(*auto split*: *if-split-asm*)

lemma *optimize-matches-simple-ruleset*: *simple-ruleset rs* ⟹ *simple-ruleset* (*optimize-matches f rs*)
by(*simp add*: *optimize-matches-def optimize-matches-option-simple-ruleset*)

definition *optimize-matches-a* :: (*action* ⇒ *'a match-expr* ⇒ *'a match-expr*) ⇒ *'a rule list* ⇒ *'a rule list* **where**
optimize-matches-a f rs = *map* (λ*r*. *Rule* (*f* (*get-action r*) (*get-match r*)) (*get-action r*)) *rs*

lemma *optimize-matches-a-simple-ruleset*: *simple-ruleset rs* ⟹ *simple-ruleset* (*optimize-matches-a f rs*)
by(*simp add*: *optimize-matches-a-def simple-ruleset-def*)

lemma *optimize-matches-a-simple-ruleset-eq*:
simple-ruleset rs ⟹ (⋀ *m a*. *a* = *Accept* ∨ *a* = *Drop* ⟹ *f1 a m* = *f2 a m*) ⟹ *optimize-matches-a f1 rs* = *optimize-matches-a f2 rs*
apply(*induction rs*)
apply(*simp add*: *optimize-matches-a-def*)
apply(*simp add*: *optimize-matches-a-def*)
apply(*simp add*: *simple-ruleset-def*)
done

lemma *optimize-matches-a-preserves*: (⋀ *r*. *r* ∈ *set rs* ⟹ *P* (*f* (*get-action r*) (*get-match r*)))
⟹ ∀ *r* ∈ *set* (*optimize-matches-a f rs*). *P* (*get-match r*)
by(*induction rs*)(*simp-all add*: *optimize-matches-a-def*)

end
theory *Semantics*
imports *Main Firewall-Common Common/List-Misc HOL−Library.LaTeXsugar*
begin

4 Big Step Semantics

The assumption we apply in general is that the firewall does not alter any packets.

A firewall ruleset is a map of chain names (e.g., INPUT, OUTPUT, FORWARD, arbitrary-user-defined-chain) to a list of rules. The list of rules is processed sequentially.

type-synonym *'a ruleset = string ⇀ 'a rule list*

A matcher (parameterized by the type of primitive *'a* and packet *'p*) is a function which just tells whether a given primitive and packet matches.

type-synonym *('a, 'p) matcher = 'a ⇒ 'p ⇒ bool*

Example: Assume a network packet only has a destination street number (for simplicity, of type *nat*) and we only support the following match expression: Is the packet's street number within a certain range? The type for the primitive could then be *nat × nat* and a possible implementation for *(nat × nat, nat) matcher* could be *match-street-number (a, b) p = (p ∈ {a..b})*. Usually, the primitives are a datatype which supports interfaces, IP addresses, protocols, ports, payload, ...

Given an *('a, 'p) matcher* and a match expression, does a packet of type *'p* match the match expression?

fun *matches* :: *('a, 'p) matcher ⇒ 'a match-expr ⇒ 'p ⇒ bool* **where**
matches γ (MatchAnd e1 e2) p ⟷ matches γ e1 p ∧ matches γ e2 p |
matches γ (MatchNot me) p ⟷ ¬ matches γ me p |
matches γ (Match e) p ⟷ γ e p |
matches - MatchAny - ⟷ True

inductive *iptables-bigstep* :: *'a ruleset ⇒ ('a, 'p) matcher ⇒ 'p ⇒ 'a rule list ⇒ state ⇒ state ⇒ bool*
(‹-,-,-⊢ ⟨-, -⟩ ⇒ -› [60,60,60,20,98,98] 89)
for *Γ* **and** *γ* **and** *p* **where**
skip: *Γ,γ,p⊢ ⟨[], t⟩ ⇒ t |*
accept: *matches γ m p ⟹ Γ,γ,p⊢ ⟨[Rule m Accept], Undecided⟩ ⇒ Decision FinalAllow |*
drop: *matches γ m p ⟹ Γ,γ,p⊢ ⟨[Rule m Drop], Undecided⟩ ⇒ Decision FinalDeny |*
reject: *matches γ m p ⟹ Γ,γ,p⊢ ⟨[Rule m Reject], Undecided⟩ ⇒ Decision FinalDeny |*
log: *matches γ m p ⟹ Γ,γ,p⊢ ⟨[Rule m Log], Undecided⟩ ⇒ Undecided |*

empty: *matches γ m p ⟹ Γ,γ,p⊢ ⟨[Rule m Empty], Undecided⟩ ⇒ Undecided |*
nomatch: *¬ matches γ m p ⟹ Γ,γ,p⊢ ⟨[Rule m a], Undecided⟩ ⇒ Undecided |*
decision: *Γ,γ,p⊢ ⟨rs, Decision X⟩ ⇒ Decision X |*
seq: *⟦Γ,γ,p⊢ ⟨rs₁, Undecided⟩ ⇒ t; Γ,γ,p⊢ ⟨rs₂, t⟩ ⇒ t'⟧ ⟹ Γ,γ,p⊢ ⟨rs₁@rs₂, Undecided⟩ ⇒ t' |*

call-return: $[\![$ *matches* γ m p; Γ *chain* = *Some* (rs_1@[*Rule* m' *Return*]@rs_2); *matches* γ m' p; Γ,γ,p⊢ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* $]\!]$ ⟹ Γ,γ,p⊢ ⟨[*Rule* m (*Call chain*)], *Undecided*⟩ ⇒ *Undecided* |
call-result: $[\![$ *matches* γ m p; Γ *chain* = *Some* rs; Γ,γ,p⊢ ⟨rs, *Undecided*⟩ ⇒ t $]\!]$ ⟹ Γ,γ,p⊢ ⟨[*Rule* m (*Call chain*)], *Undecided*⟩ ⇒ t

The semantic rules again in pretty format:

$$\frac{}{\Gamma,\gamma,p\vdash \langle [],\ t\rangle \Rightarrow t}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash \langle [\mathit{Rule}\ m\ \mathit{Accept}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalAllow}}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash \langle [\mathit{Rule}\ m\ \mathit{Drop}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalDeny}}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash \langle [\mathit{Rule}\ m\ \mathit{Reject}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalDeny}}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash \langle [\mathit{Rule}\ m\ \mathit{Log}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash \langle [\mathit{Rule}\ m\ \mathit{Empty}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}}$$

$$\frac{\neg\ \mathit{matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash \langle [\mathit{Rule}\ m\ a],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}}$$

$$\Gamma,\gamma,p\vdash \langle rs,\ \mathit{Decision}\ X\rangle \Rightarrow \mathit{Decision}\ X$$

$$\frac{\Gamma,\gamma,p\vdash \langle rs_1,\ \mathit{Undecided}\rangle \Rightarrow t \qquad \Gamma,\gamma,p\vdash \langle rs_2,\ t\rangle \Rightarrow t'}{\Gamma,\gamma,p\vdash \langle rs_1\ @\ rs_2,\ \mathit{Undecided}\rangle \Rightarrow t'}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p \qquad \Gamma\ \mathit{chain} = \mathit{Some}\ (rs_1\ @\ [\mathit{Rule}\ m'\ \mathit{Return}]\ @\ rs_2) \qquad \mathit{matches}\ \gamma\ m'\ p \qquad \Gamma,\gamma,p\vdash \langle rs_1,\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}}{\Gamma,\gamma,p\vdash \langle [\mathit{Rule}\ m\ (\mathit{Call}\ \mathit{chain})],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p \qquad \Gamma\ \mathit{chain} = \mathit{Some}\ rs \qquad \Gamma,\gamma,p\vdash \langle rs,\ \mathit{Undecided}\rangle \Rightarrow t}{\Gamma,\gamma,p\vdash \langle [\mathit{Rule}\ m\ (\mathit{Call}\ \mathit{chain})],\ \mathit{Undecided}\rangle \Rightarrow t}$$

lemma *deny*:
matches γ m p ⟹ a = *Drop* ∨ a = *Reject* ⟹ *iptables-bigstep* Γ γ p [*Rule* m a] *Undecided* (*Decision FinalDeny*)
by (*auto intro*: *drop reject*)

lemma *seq-cons*:
assumes Γ,γ,p⊢ ⟨[r],*Undecided*⟩ ⇒ t **and** Γ,γ,p⊢ ⟨rs,t⟩ ⇒ t'
shows Γ,γ,p⊢ ⟨r#rs, *Undecided*⟩ ⇒ t'

proof −
from *assms* **have** $\Gamma,\gamma,p\vdash \langle[r] @ rs, Undecided\rangle \Rightarrow t'$ **by** (*rule seq*)
thus *?thesis* **by** *simp*
qed

lemma *iptables-bigstep-induct*
[*case-names Skip Allow Deny Log Nomatch Decision Seq Call-return Call-result*,
induct pred: *iptables-bigstep*]:
$[\![\Gamma,\gamma,p\vdash \langle rs,s\rangle \Rightarrow t;$
$\bigwedge t.\ P\ []\ t\ t;$
$\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Accept \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalAllow);$
$\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Drop \vee a = Reject \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalDeny);$
$\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Log \vee a = Empty \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided;$
$\bigwedge m\ a.\ \neg\ matches\ \gamma\ m\ p \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided;$
$\bigwedge rs\ X.\ P\ rs\ (Decision\ X)\ (Decision\ X);$
$\bigwedge rs\ rs_1\ rs_2\ t\ t'.\ rs = rs_1 @ rs_2 \Longrightarrow \Gamma,\gamma,p\vdash \langle rs_1,Undecided\rangle \Rightarrow t \Longrightarrow P\ rs_1\ Undecided\ t \Longrightarrow \Gamma,\gamma,p\vdash \langle rs_2,t\rangle \Rightarrow t' \Longrightarrow P\ rs_2\ t\ t' \Longrightarrow P\ rs\ Undecided\ t';$
$\bigwedge m\ a\ chain\ rs_1\ m'\ rs_2.\ matches\ \gamma\ m\ p \Longrightarrow a = Call\ chain \Longrightarrow \Gamma\ chain = Some\ (rs_1 @ [Rule\ m'\ Return] @ rs_2) \Longrightarrow matches\ \gamma\ m'\ p \Longrightarrow \Gamma,\gamma,p\vdash \langle rs_1,Undecided\rangle \Rightarrow Undecided \Longrightarrow P\ rs_1\ Undecided\ Undecided \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided;$
$\bigwedge m\ a\ chain\ rs\ t.\ matches\ \gamma\ m\ p \Longrightarrow a = Call\ chain \Longrightarrow \Gamma\ chain = Some\ rs \Longrightarrow \Gamma,\gamma,p\vdash \langle rs,Undecided\rangle \Rightarrow t \Longrightarrow P\ rs\ Undecided\ t \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ t\ ]\!] \Longrightarrow$
$P\ rs\ s\ t$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *skipD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [] \Longrightarrow s = t$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *decisionD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow s = Decision\ X \Longrightarrow t = Decision\ X$
by (*induction rule*: *iptables-bigstep-induct*) *auto*

context
notes *skipD*[*dest*] *list-app-singletonE*[*elim*]
begin

lemma *acceptD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Accept] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Decision\ FinalAllow$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *dropD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Drop] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Decision\ FinalDeny$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *rejectD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Reject] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Decision\ FinalDeny$

by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *logD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Log] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Undecided$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *emptyD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Empty] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Undecided$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *nomatchD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ a] \Longrightarrow s = Undecided \Longrightarrow \neg\ matches\ \gamma\ m\ p \Longrightarrow t = Undecided$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *callD*:
  assumes $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t\ r = [Rule\ m\ (Call\ chain)]\ s = Undecided\ matches\ \gamma\ m\ p\ \Gamma\ chain = Some\ rs$
  obtains $\Gamma,\gamma,p\vdash \langle rs,s\rangle \Rightarrow t$
      | $rs_1\ rs_2\ m'$ **where** $rs = rs_1\ @\ Rule\ m'\ Return\ \#\ rs_2\ matches\ \gamma\ m'\ p\ \Gamma,\gamma,p\vdash \langle rs_1,s\rangle \Rightarrow Undecided\ t = Undecided$
  using *assms*
  proof (*induction r s t arbitrary*: *rs rule*: *iptables-bigstep.induct*)
    case (*seq* rs_1)
    thus *?case* **by** (*cases* rs_1) *auto*
  qed *auto*

end

lemmas *iptables-bigstepD = skipD acceptD dropD rejectD logD emptyD nomatchD decisionD callD*

lemma *seq'*:
  assumes $rs = rs_1\ @\ rs_2\ \Gamma,\gamma,p\vdash \langle rs_1,s\rangle \Rightarrow t\ \Gamma,\gamma,p\vdash \langle rs_2,t\rangle \Rightarrow t'$
  shows $\Gamma,\gamma,p\vdash \langle rs,s\rangle \Rightarrow t'$
using *assms* **by** (*cases s*) (*auto intro*: *seq decision dest*: *decisionD*)

lemma *seq'-cons*: $\Gamma,\gamma,p\vdash \langle[r],s\rangle \Rightarrow t \Longrightarrow \Gamma,\gamma,p\vdash \langle rs,t\rangle \Rightarrow t' \Longrightarrow \Gamma,\gamma,p\vdash \langle r\#rs,\ s\rangle \Rightarrow t'$
by (*metis decision decisionD state.exhaust seq-cons*)

lemma *seq-split*:
  assumes $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t\ rs = rs_1@rs_2$
  obtains t' **where** $\Gamma,\gamma,p\vdash \langle rs_1,s\rangle \Rightarrow t'\ \Gamma,\gamma,p\vdash \langle rs_2,t'\rangle \Rightarrow t$
  using *assms*
  proof (*induction rs s t arbitrary*: $rs_1\ rs_2$ *thesis rule*: *iptables-bigstep-induct*)
    case *Allow* **thus** *?case* **by** (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros*)
  next
    case *Deny* **thus** *?case* **by** (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros*)
  next

case *Log* **thus** *?case* **by** (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros*)
next
case *Nomatch* **thus** *?case* **by** (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros*)
next
case (*Seq rs rsa rsb t t'*)
hence *rs*: *rsa* @ *rsb* = rs_1 @ rs_2 **by** *simp*
note *List.append-eq-append-conv-if*[*simp*]
from *rs* **show** *?case*
proof (*cases rule*: *list-app-eq-cases*)
case *longer*
with *Seq* **have** *t1*: Γ,γ,*p*⊢ ⟨*take* (*length rsa*) rs_1, *Undecided*⟩ ⇒ *t*
by *simp*
from *Seq longer* **obtain** *t2*
where *t2a*: Γ,γ,*p*⊢ ⟨*drop* (*length rsa*) rs_1,*t*⟩ ⇒ *t2*
and *rs2-t2*: Γ,γ,*p*⊢ ⟨rs_2,*t2*⟩ ⇒ *t'*
by *blast*
with *t1 rs2-t2* **have** Γ,γ,*p*⊢ ⟨*take* (*length rsa*) rs_1 @ *drop* (*length rsa*) rs_1,*Undecided*⟩ ⇒ *t2*
by (*blast intro*: *iptables-bigstep.seq*)
with *Seq rs2-t2* **show** *?thesis*
by *simp*
next
case *shorter*
with *rs* **have** *rsa'*: *rsa* = rs_1 @ *take* (*length rsa* − *length* rs_1) rs_2
by (*metis append-eq-conv-conj length-drop*)
from *shorter rs* **have** *rsb'*: *rsb* = *drop* (*length rsa* − *length* rs_1) rs_2
by (*metis append-eq-conv-conj length-drop*)
from *Seq rsa'* **obtain** *t1*
where *t1a*: Γ,γ,*p*⊢ ⟨rs_1,*Undecided*⟩ ⇒ *t1*
and *t1b*: Γ,γ,*p*⊢ ⟨*take* (*length rsa* − *length* rs_1) rs_2,*t1*⟩ ⇒ *t*
by *blast*
from *rsb' Seq.hyps* **have** *t2*: Γ,γ,*p*⊢ ⟨*drop* (*length rsa* − *length* rs_1) rs_2,*t*⟩ ⇒ *t'*
by *blast*
with *seq' t1b* **have** Γ,γ,*p*⊢ ⟨rs_2,*t1*⟩ ⇒ *t'*
by *fastforce*
with *Seq t1a* **show** *?thesis*
by *fast*
qed
next
case *Call-return*
hence Γ,γ,*p*⊢ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* Γ,γ,*p*⊢ ⟨rs_2, *Undecided*⟩ ⇒ *Undecided*
by (*case-tac* [!] rs_1) (*auto intro*: *iptables-bigstep.skip iptables-bigstep.call-return*)
thus *?case* **by** *fact*
next
case (*Call-result - - - - t*)
show *?case*
proof (*cases* rs_1)

```
      case Nil
       with Call-result have Γ,γ,p⊢ ⟨rs₁, Undecided⟩ ⇒ Undecided Γ,γ,p⊢ ⟨rs₂,
Undecided⟩ ⇒ t
        by (auto intro: iptables-bigstep.intros)
      thus ?thesis by fact
    next
      case Cons
      with Call-result have Γ,γ,p⊢ ⟨rs₁, Undecided⟩ ⇒ t Γ,γ,p⊢ ⟨rs₂, t⟩ ⇒ t
        by (auto intro: iptables-bigstep.intros)
      thus ?thesis by fact
    qed
  qed (auto intro: iptables-bigstep.intros)

lemma seqE:
  assumes Γ,γ,p⊢ ⟨rs₁@rs₂, s⟩ ⇒ t
  obtains ti where Γ,γ,p⊢ ⟨rs₁,s⟩ ⇒ ti Γ,γ,p⊢ ⟨rs₂,ti⟩ ⇒ t
  using assms by (force elim: seq-split)

lemma seqE-cons:
  assumes Γ,γ,p⊢ ⟨r#rs, s⟩ ⇒ t
  obtains ti where Γ,γ,p⊢ ⟨[r],s⟩ ⇒ ti Γ,γ,p⊢ ⟨rs,ti⟩ ⇒ t
  using assms by (metis append-Cons append-Nil seqE)

lemma nomatch':
  assumes ⋀r. r ∈ set rs ⟹ ¬ matches γ (get-match r) p
  shows Γ,γ,p⊢ ⟨rs, s⟩ ⇒ s
  proof(cases s)
    case Undecided
     have ∀ r∈set rs. ¬ matches γ (get-match r) p ⟹ Γ,γ,p⊢ ⟨rs, Undecided⟩ ⇒
Undecided
      proof(induction rs)
        case Nil
        thus ?case by (fast intro: skip)
      next
        case (Cons r rs)
        hence Γ,γ,p⊢ ⟨[r], Undecided⟩ ⇒ Undecided
          by (cases r) (auto intro: nomatch)
        with Cons show ?case
          by (fastforce intro: seq-cons)
      qed
    with assms Undecided show ?thesis by simp
  qed (blast intro: decision)
```

there are only two cases when there can be a Return on top-level:

- the firewall is in a Decision state
- the return does not match

In both cases, it is not applied!

lemma *no-free-return*: **assumes** $\Gamma,\gamma,p\vdash \langle[Rule\ m\ Return], Undecided\rangle \Rightarrow t$ **and** *matches* $\gamma\ m\ p$ **shows** *False*
proof −
{ fix *a s*
have *no-free-return-hlp*: $\Gamma,\gamma,p\vdash \langle a,s\rangle \Rightarrow t \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow a = [Rule\ m\ Return] \Longrightarrow False$
proof (*induction rule*: *iptables-bigstep.induct*)
case (*seq* rs_1)
thus *?case*
by (*cases* rs_1) (*auto dest*: *skipD*)
qed *simp-all*
} with *assms* **show** *?thesis* **by** *blast*
qed

lemma *seq-progress*: $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t \Longrightarrow rs = rs_1@rs_2 \Longrightarrow \Gamma,\gamma,p\vdash \langle rs_1,\ s\rangle \Rightarrow t' \Longrightarrow \Gamma,\gamma,p\vdash \langle rs_2,\ t'\rangle \Rightarrow t$
proof(*induction arbitrary*: $rs_1\ rs_2\ t'$ *rule*: *iptables-bigstep-induct*)
case *Allow*
thus *?case*
by (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros dest*: *iptables-bigstepD*)
next
case *Deny*
thus *?case*
by (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros dest*: *iptables-bigstepD*)
next
case *Log*
thus *?case*
by (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros dest*: *iptables-bigstepD*)
next
case *Nomatch*
thus *?case*
by (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros dest*: *iptables-bigstepD*)
next
case *Decision*
thus *?case*
by (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros dest*: *iptables-bigstepD*)
next
case(*Seq rs rsa rsb t t'* $rs_1\ rs_2\ t''$)
hence *rs*: *rsa* @ *rsb* = rs_1 @ rs_2 **by** *simp*
note *List.append-eq-append-conv-if*[*simp*]

from *rs* **show** $\Gamma,\gamma,p\vdash \langle rs_2,t''\rangle \Rightarrow t'$
proof(*cases rule*: *list-app-eq-cases*)
case *longer*
have rs_1 = *take* (*length rsa*) rs_1 @ *drop* (*length rsa*) rs_1
by *auto*

```
      with Seq longer show ?thesis
        by (metis append-Nil2 skipD seq-split)
    next
      case shorter
      with Seq(7) Seq.hyps(3) Seq.IH(1) rs show ?thesis
        by (metis seq' append-eq-conv-conj)
    qed
  next
    case(Call-return m a chain rsa m' rsb)
    have xx: Γ,γ,p⊢ ⟨[Rule m (Call chain)], Undecided⟩ ⇒ t' ⟹ matches γ m p
⟹
        Γ chain = Some (rsa @ Rule m' Return # rsb) ⟹
        matches γ m' p ⟹
        Γ,γ,p⊢ ⟨rsa, Undecided⟩ ⇒ Undecided ⟹
        t' = Undecided
      apply(erule callD)
       apply(simp-all)
      apply(erule seqE)
      apply(erule seqE-cons)
      by (metis Call-return.IH no-free-return self-append-conv skipD)

    show ?case
      proof (cases rs1)
        case (Cons r rs)
        thus ?thesis
          using Call-return
          apply(case-tac [Rule m a] = rs2)
           apply(simp)
          apply(simp)
          using xx by blast
      next
        case Nil
        moreover hence t' = Undecided
            by (metis Call-return.hyps(1) Call-return.prems(2) append.simps(1)
decision no-free-return seq state.exhaust)
        moreover have ⋀m. Γ,γ,p⊢ ⟨[Rule m a], Undecided⟩ ⇒ Undecided
        by (metis (no-types) Call-return(2) Call-return.hyps(3) Call-return.hyps(4)
Call-return.hyps(5) call-return nomatch)
        ultimately show ?thesis
          using Call-return.prems(1) by auto
      qed
  next
    case(Call-result m a chain rs t)
    thus ?case
      proof (cases rs1)
        case Cons
        thus ?thesis
          using Call-result
          apply(auto simp add: iptables-bigstep.skip iptables-bigstep.call-result dest:
```

skipD)
apply(*drule callD*, *simp-all*)
apply *blast*
by (*metis Cons-eq-appendI append-self-conv2 no-free-return seq-split*)
qed (*fastforce intro*: *iptables-bigstep.intros dest*: *skipD*)
qed (*auto dest*: *iptables-bigstepD*)

theorem *iptables-bigstep-deterministic*: **assumes** Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t* **and** Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t'* **shows** *t* = *t'*
proof −
{ fix *r1 r2 m t*
assume *a1*: Γ,γ,*p*⊢ ⟨*r1* @ *Rule m Return* # *r2*, *Undecided*⟩ ⇒ *t* **and** *a2*: *matches* γ *m p* **and** *a3*: Γ,γ,*p*⊢ ⟨*r1*,*Undecided*⟩ ⇒ *Undecided*
have *False*
proof −
from *a1 a3* **have** Γ,γ,*p*⊢ ⟨*Rule m Return* # *r2*, *Undecided*⟩ ⇒ *t*
by (*blast intro*: *seq-progress*)
hence Γ,γ,*p*⊢ ⟨[*Rule m Return*] @ *r2*, *Undecided*⟩ ⇒ *t*
by *simp*
from *seqE*[*OF this*] **obtain** *ti* **where** Γ,γ,*p*⊢ ⟨[*Rule m Return*], *Undecided*⟩ ⇒ *ti* **by** *blast*
with *no-free-return a2* **show** *False* **by** *fast*
qed
} note *no-free-return-seq*=*this*

from *assms* **show** *?thesis*
proof (*induction arbitrary*: *t'* *rule*: *iptables-bigstep-induct*)
case *Seq*
thus *?case*
by (*metis seq-progress*)
next
case *Call-result*
thus *?case*
by (*metis no-free-return-seq callD*)
next
case *Call-return*
thus *?case*
by (*metis append-Cons callD no-free-return-seq*)
qed (*auto dest*: *iptables-bigstepD*)
qed

lemma *iptables-bigstep-to-undecided*: Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *Undecided* ⟹ *s* = *Undecided*
by (*metis decisionD state.exhaust*)

lemma *iptables-bigstep-to-decision*: Γ,γ,*p*⊢ ⟨*rs*, *Decision Y*⟩ ⇒ *Decision X* ⟹ *Y* = *X*
by (*metis decisionD state.inject*)

lemma *Rule-UndecidedE*:
assumes $\Gamma,\gamma,p\vdash \langle[Rule\ m\ a],\ Undecided\rangle \Rightarrow Undecided$
obtains (*nomatch*) $\neg\ matches\ \gamma\ m\ p$
| (*log*) $a = Log \vee a = Empty$
| (*call*) *c* **where** $a = Call\ c\ matches\ \gamma\ m\ p$
using *assms*
proof (*induction* [*Rule m a*] *Undecided Undecided rule*: *iptables-bigstep-induct*)
case *Seq*
thus *?case*
by (*metis append-eq-Cons-conv append-is-Nil-conv iptables-bigstep-to-undecided*)
qed *simp-all*

lemma *Rule-DecisionE*:
assumes $\Gamma,\gamma,p\vdash \langle[Rule\ m\ a],\ Undecided\rangle \Rightarrow Decision\ X$
obtains (*call*) *chain* **where** $matches\ \gamma\ m\ p\ a = Call\ chain$
| (*accept-reject*) $matches\ \gamma\ m\ p\ X = FinalAllow \Longrightarrow a = Accept\ X = FinalDeny \Longrightarrow a = Drop \vee a = Reject$
using *assms*
proof (*induction* [*Rule m a*] *Undecided Decision X rule*: *iptables-bigstep-induct*)
case (*Seq* rs_1)
thus *?case*
by (*cases* rs_1) (*auto dest*: *skipD*)
qed *simp-all*

lemma *log-remove*:
assumes $\Gamma,\gamma,p\vdash \langle rs_1 @ [Rule\ m\ Log] @ rs_2,\ s\rangle \Rightarrow t$
shows $\Gamma,\gamma,p\vdash \langle rs_1 @ rs_2,\ s\rangle \Rightarrow t$
proof −
from *assms* **obtain** t' **where** t': $\Gamma,\gamma,p\vdash \langle rs_1,\ s\rangle \Rightarrow t'\ \Gamma,\gamma,p\vdash \langle[Rule\ m\ Log] @ rs_2,\ t'\rangle \Rightarrow t$
by (*blast elim*: *seqE*)
hence $\Gamma,\gamma,p\vdash \langle Rule\ m\ Log\ \#\ rs_2,\ t'\rangle \Rightarrow t$
by *simp*
then obtain t'' **where** $\Gamma,\gamma,p\vdash \langle[Rule\ m\ Log],\ t'\rangle \Rightarrow t''\ \Gamma,\gamma,p\vdash \langle rs_2,\ t''\rangle \Rightarrow t$
by (*blast elim*: *seqE-cons*)
with t' **show** *?thesis*
by (*metis state.exhaust iptables-bigstep-deterministic decision log nomatch seq*)
qed
lemma *empty-empty*:
assumes $\Gamma,\gamma,p\vdash \langle rs_1 @ [Rule\ m\ Empty] @ rs_2,\ s\rangle \Rightarrow t$
shows $\Gamma,\gamma,p\vdash \langle rs_1 @ rs_2,\ s\rangle \Rightarrow t$
proof −
from *assms* **obtain** t' **where** t': $\Gamma,\gamma,p\vdash \langle rs_1,\ s\rangle \Rightarrow t'\ \Gamma,\gamma,p\vdash \langle[Rule\ m\ Empty] @ rs_2,\ t'\rangle \Rightarrow t$
by (*blast elim*: *seqE*)
hence $\Gamma,\gamma,p\vdash \langle Rule\ m\ Empty\ \#\ rs_2,\ t'\rangle \Rightarrow t$
by *simp*

then obtain t'' **where** $\Gamma,\gamma,p\vdash \langle[Rule\ m\ Empty],\ t'\rangle \Rightarrow t''\ \Gamma,\gamma,p\vdash \langle rs_2,\ t''\rangle \Rightarrow t$
by (*blast elim*: *seqE-cons*)
with t' **show** *?thesis*
by (*metis state.exhaust iptables-bigstep-deterministic decision empty nomatch seq*)
qed

lemma *Unknown-actions-False*: $\Gamma,\gamma,p\vdash \langle r\ \#\ rs,\ Undecided\rangle \Rightarrow t \Longrightarrow r = Rule\ m\ a \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow a = Unknown \vee (\exists chain.\ a = Goto\ chain) \Longrightarrow False$
proof −
have *1*: $\Gamma,\gamma,p\vdash \langle[Rule\ m\ Unknown],\ Undecided\rangle \Rightarrow t \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow False$
by (*induction* [*Rule m Unknown*] *Undecided t rule*: *iptables-bigstep.induct*)
(*auto elim*: *list-app-singletonE dest*: *skipD*)

{ fix *chain*
have $\Gamma,\gamma,p\vdash \langle[Rule\ m\ (Goto\ chain)],\ Undecided\rangle \Rightarrow t \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow False$
by (*induction* [*Rule m* (*Goto chain*)] *Undecided t rule*: *iptables-bigstep.induct*)
(*auto elim*: *list-app-singletonE dest*: *skipD*)
}note *2=this*
show $\Gamma,\gamma,p\vdash \langle r\ \#\ rs,\ Undecided\rangle \Rightarrow t \Longrightarrow r = Rule\ m\ a \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow a = Unknown \vee (\exists chain.\ a = Goto\ chain) \Longrightarrow False$
apply(*erule seqE-cons*)
apply(*case-tac ti*)
apply(*simp-all*)
using *Rule-UndecidedE* **apply** *fastforce*
by (*metis 1 2 decision iptables-bigstep-deterministic*)
qed

The notation we prefer in the paper. The semantics are defined for fixed Γ and γ

locale *iptables-bigstep-fixedbackground* =
fixes Γ::*'a ruleset*
and γ::(*'a*, *'p*) *matcher*
begin

inductive *iptables-bigstep'* :: *'p* ⇒ *'a rule list* ⇒ *state* ⇒ *state* ⇒ *bool*
(‹-⊢'' ⟨-, -⟩ ⇒ -› [*60*,*20*,*98*,*98*] *89*)
for *p* **where**
skip: *p*⊢' ⟨[], *t*⟩ ⇒ *t* |
accept: *matches* γ *m p* ⟹ *p*⊢' ⟨[*Rule m Accept*], *Undecided*⟩ ⇒ *Decision FinalAllow* |
drop: *matches* γ *m p* ⟹ *p*⊢' ⟨[*Rule m Drop*], *Undecided*⟩ ⇒ *Decision FinalDeny* |
reject: *matches* γ *m p* ⟹ *p*⊢' ⟨[*Rule m Reject*], *Undecided*⟩ ⇒ *Decision FinalDeny* |

log: $matches\ \gamma\ m\ p \Longrightarrow p\vdash' \langle[Rule\ m\ Log], Undecided\rangle \Rightarrow Undecided$ |
empty: $matches\ \gamma\ m\ p \Longrightarrow p\vdash' \langle[Rule\ m\ Empty], Undecided\rangle \Rightarrow Undecided$ |
nomatch: $\neg\ matches\ \gamma\ m\ p \Longrightarrow p\vdash' \langle[Rule\ m\ a], Undecided\rangle \Rightarrow Undecided$ |
decision: $p\vdash' \langle rs, Decision\ X\rangle \Rightarrow Decision\ X$ |
seq: $[\![p\vdash' \langle rs_1, Undecided\rangle \Rightarrow t; p\vdash' \langle rs_2, t\rangle \Rightarrow t']\!] \Longrightarrow p\vdash' \langle rs_1@rs_2, Undecided\rangle \Rightarrow t'$ |
call-return: $[\![\ matches\ \gamma\ m\ p; \Gamma\ chain = Some\ (rs_1@[Rule\ m'\ Return]@rs_2); matches\ \gamma\ m'\ p; p\vdash' \langle rs_1, Undecided\rangle \Rightarrow Undecided\ ]\!] \Longrightarrow p\vdash' \langle[Rule\ m\ (Call\ chain)], Undecided\rangle \Rightarrow Undecided$ |
call-result: $[\![\ matches\ \gamma\ m\ p; p\vdash' \langle the\ (\Gamma\ chain), Undecided\rangle \Rightarrow t\ ]\!] \Longrightarrow p\vdash' \langle[Rule\ m\ (Call\ chain)], Undecided\rangle \Rightarrow t$

definition *wf-Γ*:: *'a rule list* ⇒ *bool* **where**
wf-Γ rs ≡ ∀ *rsg* ∈ *ran Γ* ∪ {*rs*}. (∀ *r* ∈ *set rsg*. ∀ *chain*. *get-action r* = *Call chain* ⟶ *Γ chain* ≠ *None*)

lemma *wf-Γ-append*: *wf-Γ* (*rs1*@*rs2*) ⟷ *wf-Γ rs1* ∧ *wf-Γ rs2*
by(*simp add*: *wf-Γ-def*, *blast*)
lemma *wf-Γ-tail*: *wf-Γ* (*r* # *rs*) ⟹ *wf-Γ rs* **by**(*simp add*: *wf-Γ-def*)
lemma *wf-Γ-Call*: *wf-Γ* [*Rule m* (*Call chain*)] ⟹ *wf-Γ* (*the* (*Γ chain*)) ∧ (∃ *rs*. *Γ chain* = *Some rs*)
apply(*simp add*: *wf-Γ-def*)
by (*metis option.collapse ranI*)

lemma *wf-Γ rs* ⟹ *p*⊢' ⟨*rs*, *s*⟩ ⇒ *t* ⟷ *Γ*,*γ*,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t*
apply(*rule iffI*)
apply(*rotate-tac 1*)
apply(*induction rs s t rule*: *iptables-bigstep'.induct*)
apply(*auto intro*: *iptables-bigstep.intros simp*: *wf-Γ-append dest*!: *wf-Γ-Call*)[*11*]
apply(*rotate-tac 1*)
apply(*induction rs s t rule*: *iptables-bigstep.induct*)
apply(*auto intro*: *iptables-bigstep'.intros simp*: *wf-Γ-append dest*!: *wf-Γ-Call*)[*11*]
done

end

Showing that semantics are defined. For rulesets which can be loaded by the Linux kernel. The kernel does not allow loops.

We call a ruleset well-formed (wf) iff all *Call*s are into actually existing chains.

definition *wf-chain* :: *'a ruleset* ⇒ *'a rule list* ⇒ *bool* **where**
wf-chain Γ rs ≡ (∀ *r* ∈ *set rs*. ∀ *chain*. *get-action r* = *Call chain* ⟶ *Γ chain* ≠ *None*)
lemma *wf-chain-append*: *wf-chain Γ* (*rs1*@*rs2*) ⟷ *wf-chain Γ rs1* ∧ *wf-chain Γ rs2*
by(*simp add*: *wf-chain-def*, *blast*)

lemma *wf-chain-fst*: *wf-chain* Γ (*r* # *rs*) ⟹ *wf-chain* Γ (*rs*)
by(*simp add*: *wf-chain-def*)

This is what our tool will check at runtime

definition *sanity-wf-ruleset* :: (*string* × *'a rule list*) *list* ⇒ *bool* **where**
sanity-wf-ruleset Γ ≡ *distinct* (*map fst* Γ) ∧
(∀ *rs* ∈ *ran* (*map-of* Γ). (∀ *r* ∈ *set rs*. *case get-action r of Accept* ⇒ *True*
| *Drop* ⇒ *True*
| *Reject* ⇒ *True*
| *Log* ⇒ *True*
| *Empty* ⇒ *True*
| *Call chain* ⇒ *chain* ∈ *dom* (*map-of* Γ)
| *Goto chain* ⇒ *chain* ∈ *dom* (*map-of* Γ)
| *Return* ⇒ *True*
| - ⇒ *False*))

lemma *sanity-wf-ruleset-wf-chain*: *sanity-wf-ruleset* Γ ⟹ *rs* ∈ *ran* (*map-of* Γ) ⟹ *wf-chain* (*map-of* Γ) *rs*
apply(*simp add*: *sanity-wf-ruleset-def wf-chain-def*)
by *fastforce*

lemma *sanity-wf-ruleset-start*: *sanity-wf-ruleset* Γ ⟹ *chain-name* ∈ *dom* (*map-of* Γ) ⟹
default-action = *Accept* ∨ *default-action* = *Drop* ⟹
wf-chain (*map-of* Γ) [*Rule MatchAny* (*Call chain-name*), *Rule MatchAny default-action*]
apply(*simp add*: *sanity-wf-ruleset-def wf-chain-def*)
apply(*safe*)
apply(*simp-all*)
apply *blast+*
done

lemma [*code*]: *sanity-wf-ruleset* Γ =
(*let dom* = *map fst* Γ;
ran = *map snd* Γ
in distinct dom ∧
(∀ *rs* ∈ *set ran*. (∀ *r* ∈ *set rs*. *case get-action r of Accept* ⇒ *True*
| *Drop* ⇒ *True*
| *Reject* ⇒ *True*
| *Log* ⇒ *True*
| *Empty* ⇒ *True*
| *Call chain* ⇒ *chain* ∈ *set dom*
| *Goto chain* ⇒ *chain* ∈ *set dom*
| *Return* ⇒ *True*
| - ⇒ *False*)))
proof −

```
  have set-map-fst: set (map fst Γ) = dom (map-of Γ)
    by (simp add: dom-map-of-conv-image-fst)
  have set-map-snd: distinct (map fst Γ) ⟹ set (map snd Γ) = ran (map-of Γ)
    by (simp add: ran-distinct)
  show ?thesis
  unfolding sanity-wf-ruleset-def Let-def
  apply(subst set-map-fst)+
  apply(rule iffI)
   apply(elim conjE)
   apply(subst set-map-snd)
    apply(simp)
   apply(simp)
  apply(elim conjE)
  apply(subst(asm) set-map-snd)
   apply(simp-all)
  done
qed
```

```
lemma semantics-bigstep-defined1: assumes ∀ rsg ∈ ran Γ ∪ {rs}. wf-chain Γ rsg
  and ∀ rsg ∈ ran Γ ∪ {rs}. ∀  r ∈ set rsg. (∀ chain. get-action r ≠ Goto chain) ∧
get-action r ≠ Unknown
  and ∀  r ∈ set rs. get-action r ≠ Return
  and (∀ name ∈ dom Γ. ∃ t. Γ,γ,p⊢ ⟨the (Γ name), Undecided⟩ ⇒ t)
  shows ∃ t. Γ,γ,p⊢ ⟨rs, s⟩ ⇒ t
using assms proof(induction rs)
case Nil thus ?case
 apply(rule-tac x=s in exI)
 by(simp add: skip)
next
case (Cons r rs)
  from Cons.prems Cons.IH obtain t' where t': Γ,γ,p⊢ ⟨rs, s⟩ ⇒ t'
    apply simp
    apply(elim conjE)
    apply(simp add: wf-chain-fst)
    by blast

  obtain m a where r: r = Rule m a by(cases r) blast

  show ?case
  proof(cases matches γ m p)
  case False
    hence Γ,γ,p⊢ ⟨[r], s⟩ ⇒ s
      apply(cases s)
       apply(simp add: nomatch r)
      by(simp add: decision)
```

```
    thus ?thesis
      apply(rule-tac x=t' in exI)
      apply(rule-tac t=s in seq'-cons)
       apply assumption
      using t' by(simp)
  next
  case True
    show ?thesis
    proof(cases s)
    case (Decision X) thus ?thesis
      apply(rule-tac x=Decision X in exI)
      by(simp add: decision)
    next
    case Undecided
      have ∃ t. Γ,γ,p⊢ ⟨Rule m a # rs, Undecided⟩ ⇒ t
      proof(cases a)
        case Accept with True show ?thesis
          apply(rule-tac x=Decision FinalAllow in exI)
          apply(rule-tac t=Decision FinalAllow in seq'-cons)
           by(auto intro: iptables-bigstep.intros)
        next
        case Drop with True show ?thesis
          apply(rule-tac x=Decision FinalDeny in exI)
          apply(rule-tac t=Decision FinalDeny in seq'-cons)
           by(auto intro: iptables-bigstep.intros)
        next
        case Log with True t' Undecided show ?thesis
          apply(rule-tac x=t' in exI)
          apply(rule-tac t=Undecided in seq'-cons)
           by(auto intro: iptables-bigstep.intros)
        next
        case Reject with True show ?thesis
          apply(rule-tac x=Decision FinalDeny in exI)
          apply(rule-tac t=Decision FinalDeny in seq'-cons)
           by(auto intro: iptables-bigstep.intros)[2]
        next
        case Return with Cons.prems(3)[simplified r] show ?thesis by simp
        next
        case Goto with Cons.prems(2)[simplified r] show ?thesis by auto
        next
        case (Call chain-name)
         from Call Cons.prems(1) obtain rs' where 1: Γ chain-name = Some rs'
by(simp add: r wf-chain-def) blast
          with Cons.prems(4) obtain t'' where 2: Γ,γ,p⊢ ⟨the (Γ chain-name),
Undecided⟩ ⇒ t'' by blast
          from 1 2 True have Γ,γ,p⊢ ⟨[Rule m (Call chain-name)], Undecided⟩ ⇒
t'' by(auto dest: call-result)
          with Call t' Undecided show ?thesis
          apply(simp add: r)
```

```
        apply(cases t'')
         apply simp
         apply(rule-tac x=t' in exI)
         apply(rule-tac t=Undecided in seq'-cons)
          apply(auto intro: iptables-bigstep.intros)[2]
        apply(simp)
        apply(rule-tac x=t'' in exI)
        apply(rule-tac t=t'' in seq'-cons)
         apply(auto intro: iptables-bigstep.intros)
       done
      next
      case Empty  with True t' Undecided show ?thesis
       apply(rule-tac x=t' in exI)
       apply(rule-tac t=Undecided in seq'-cons)
        by(auto intro: iptables-bigstep.intros)
      next
      case Unknown with Cons.prems(2)[simplified r] show ?thesis by(simp)
    qed
    thus ?thesis
    unfolding r Undecided by simp
  qed
 qed
qed
```

Showing the main theorem

```
context
begin
 private lemma iptables-bigstep-defined-if-singleton-rules:
 ∀ r ∈ set rs. (∃ t. Γ,γ,p⊢ ⟨[r], s⟩ ⇒ t) ⟹ ∃ t. Γ,γ,p⊢ ⟨rs, s⟩ ⇒ t
 proof(induction rs arbitrary: s)
 case Nil hence Γ,γ,p⊢ ⟨[], s⟩ ⇒ s by(simp add: skip)
   thus ?case by blast
 next
 case(Cons r rs s)
  from Cons.prems obtain t where t: Γ,γ,p⊢ ⟨[r], s⟩ ⇒ t by simp blast
  with Cons show ?case
  proof(cases t)
   case Decision with t show ?thesis by (meson decision seq'-cons)
   next
   case Undecided
   from Cons obtain t' where t': Γ,γ,p⊢ ⟨rs, s⟩ ⇒ t' by simp blast
   with Undecided t show ?thesis
   apply(rule-tac x=t' in exI)
   apply(rule seq'-cons)
    apply(simp)
   using iptables-bigstep-to-undecided by fastforce
  qed
 qed
```

well founded relation.

definition *calls-chain* :: *'a ruleset* ⇒ (*string* × *string*) *set* **where**
calls-chain Γ = {(*r*, *s*). *case* Γ *r of Some rs* ⇒ ∃ *m*. *Rule m* (*Call s*) ∈ *set rs* | *None* ⇒ *False*}

lemma *calls-chain-def2*: *calls-chain* Γ = {(*caller*, *callee*). ∃ *rs m*. Γ *caller* = *Some rs* ∧ *Rule m* (*Call callee*) ∈ *set rs*}
unfolding *calls-chain-def*
apply(*safe*)
apply(*simp split*: *option.split-asm*)
apply(*simp*)
by *blast*

example

private lemma *calls-chain* [
''FORWARD'' ↦ [(*Rule m1 Log*), (*Rule m2* (*Call ''foo''*)), (*Rule m3 Accept*), (*Rule m'* (*Call ''baz''*))],
''foo'' ↦ [(*Rule m4 Log*), (*Rule m5 Return*), (*Rule m6* (*Call ''bar''*))],
''bar'' ↦ [],
''baz'' ↦ []] =
{(*''FORWARD''*, *''foo''*), (*''FORWARD''*, *''baz''*), (*''foo''*, *''bar''*)}
unfolding *calls-chain-def* **by**(*auto split*: *option.split-asm if-split-asm*)

private lemma *wf* (*calls-chain* [
''FORWARD'' ↦ [(*Rule m1 Log*), (*Rule m2* (*Call ''foo''*)), (*Rule m3 Accept*), (*Rule m'* (*Call ''baz''*))],
''foo'' ↦ [(*Rule m4 Log*), (*Rule m5 Return*), (*Rule m6* (*Call ''bar''*))],
''bar'' ↦ [],
''baz'' ↦ []])
proof −
have *g*: *calls-chain* [*''FORWARD''* ↦ [(*Rule m1 Log*), (*Rule m2* (*Call ''foo''*)), (*Rule m3 Accept*), (*Rule m'* (*Call ''baz''*))],
''foo'' ↦ [(*Rule m4 Log*), (*Rule m5 Return*), (*Rule m6* (*Call ''bar''*))],
''bar'' ↦ [],
''baz'' ↦ [] = {(*''FORWARD''*, *''foo''*), (*''FORWARD''*, *''baz''*), (*''foo''*, *''bar''*)}
by(*auto simp add*: *calls-chain-def split*: *option.split-asm if-split-asm*)
show *?thesis*
unfolding *g*
apply(*simp*)
apply *safe*
apply(*erule rtranclE*, *simp-all*)
apply(*erule rtranclE*, *simp-all*)
done
qed

In our proof, we will need the reverse.

private definition *called-by-chain* :: *'a ruleset* ⇒ (*string* × *string*) *set* **where**
called-by-chain Γ = {(*callee*, *caller*). *case* Γ *caller of Some rs* ⇒ ∃ *m*. *Rule m* (*Call callee*) ∈ *set rs* | *None* ⇒ *False*}

private lemma *called-by-chain-converse*: *calls-chain* Γ = *converse* (*called-by-chain* Γ)
apply(*simp add*: *calls-chain-def called-by-chain-def*)
by *blast*
private lemma *wf-called-by-chain*: *finite* (*calls-chain* Γ) ⟹ *wf* (*calls-chain* Γ) ⟹ *wf* (*called-by-chain* Γ)
apply(*frule Wellfounded.wf-acyclic*)
apply(*drule*(*1*) *Wellfounded.finite-acyclic-wf-converse*)
apply(*simp add*: *called-by-chain-converse*)
done

private lemma *helper-cases-call-subchain-defined-or-return*:
(∀ *x*∈*ran* Γ. *wf-chain* Γ *x*) ⟹
∀ *rsg*∈*ran* Γ. ∀ *r*∈*set rsg*. (∀ *chain*. *get-action r* ≠ *Goto chain*) ∧ *get-action r* ≠ *Unknown* ⟹
∀ *y m*. ∀ *r*∈*set rs-called*. *r* = *Rule m* (*Call y*) ⟶ (∃ *t*. Γ,γ,*p*⊢ ⟨[*Rule m* (*Call y*)], *Undecided*⟩ ⇒ *t*) ⟹
wf-chain Γ *rs-called* ⟹
∀ *r*∈*set rs-called*. (∀ *chain*. *get-action r* ≠ *Goto chain*) ∧ *get-action r* ≠ *Unknown* ⟹
(∃ *t*. Γ,γ,*p*⊢ ⟨*rs-called*, *Undecided*⟩ ⇒ *t*) ∨
(∃ *rs-called1 rs-called2 m'*.
rs-called = (*rs-called1* @ [*Rule m' Return*] @ *rs-called2*) ∧
matches γ *m' p* ∧ Γ,γ,*p*⊢ ⟨*rs-called1*, *Undecided*⟩ ⇒ *Undecided*)
proof(*induction rs-called arbitrary*:)
case *Nil* **hence** ∃ *t*. Γ,γ,*p*⊢ ⟨[], *Undecided*⟩ ⇒ *t*
apply(*rule-tac x*=*Undecided* **in** *exI*)
by(*simp add*: *skip*)
thus *?case* **by** *simp*
next
case (*Cons r rs*)
from *Cons.prems* **have** *wf-chain* Γ [*r*] **by**(*simp add*: *wf-chain-def*)
from *Cons.prems* **have** *IH*:(∃ *t'*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *t'*) ∨
(∃ *rs-called1 rs-called2 m'*.
rs = (*rs-called1* @ [*Rule m' Return*] @ *rs-called2*) ∧
matches γ *m' p* ∧ Γ,γ,*p*⊢ ⟨*rs-called1*, *Undecided*⟩ ⇒ *Undecided*)
apply −
apply(*rule Cons.IH*)
apply(*auto dest*: *wf-chain-fst*)
done

from *Cons.prems* **have** *case-call*: *r* = *Rule m* (*Call y*) ⟹ (∃ *t*. Γ,γ,*p*⊢ ⟨[*Rule m* (*Call y*)], *Undecided*⟩ ⇒ *t*) **for** *y m*
by(*simp*)

obtain *m a* **where** *r*: *r* = *Rule m a* **by**(*cases r*) *simp*

from *Cons.prems* **have** *a-not*: (∀ *chain*. *a* ≠ *Goto chain*) ∧ *a* ≠ *Unknown*

by(*simp add*: *r*)

have *ex-neq-ret*: $a \neq Return \Longrightarrow \exists\, t.\ \Gamma,\gamma,p\vdash \langle[Rule\ m\ a],\ Undecided\rangle \Rightarrow t$
proof(*cases matches* γ *m p*)
case *False* **thus** *?thesis* **by**(*rule-tac x=Undecided* **in** *exI*)(*simp add*: *nomatch*; *fail*)
next
case *True*
assume $a \neq Return$
show *?thesis*
proof(*cases a*)
case *Accept* **with** *True* **show** *?thesis*
by(*rule-tac x=Decision FinalAllow* **in** *exI*) (*simp add*: *accept*; *fail*)
next
case *Drop* **with** *True* **show** *?thesis*
by(*rule-tac x=Decision FinalDeny* **in** *exI*) (*simp add*: *drop*; *fail*)
next
case *Log* **with** *True* **show** *?thesis*
by(*rule-tac x=Undecided* **in** *exI*)(*simp add*: *log*; *fail*)
next
case *Reject* **with** *True* **show** *?thesis*
by(*rule-tac x=Decision FinalDeny* **in** *exI*) (*simp add*: *reject*; *fail*)
next
case *Call* **with** *True* **show** *?thesis*
apply(*simp*)
apply(*rule case-call*)
apply(*simp add*: *r*; *fail*)
done
next
case *Empty* **with** *True* **show** *?thesis* **by**(*rule-tac x=Undecided* **in** *exI*) (*simp add*: *empty*; *fail*)
next
case *Return* **with** ‹$a \neq Return$› **show** *?thesis* **by** *simp*
qed(*simp-all add*: *a-not*)
qed

have $*$: *?case*
if *pre*: $rs = rs\text{-}called1\ @\ Rule\ m'\ Return\ \#\ rs\text{-}called2 \wedge matches\ \gamma\ m'\ p \wedge \Gamma,\gamma,p\vdash \langle rs\text{-}called1,\ Undecided\rangle \Rightarrow Undecided$
for *rs-called1* m' *rs-called2*
proof(*cases matches* γ *m p*)
case *False* **thus** *?thesis*
apply −
apply(*rule disjI2*)
apply(*rule-tac x=r#rs-called1* **in** *exI*)
apply(*rule-tac x=rs-called2* **in** *exI*)
apply(*rule-tac x=*m' **in** *exI*)
apply(*simp add*: *r pre*)
apply(*rule-tac t=Undecided* **in** *seq-cons*)

```
     apply(simp add: r nomatch; fail)
    apply(simp add: pre; fail)
    done
  next
  case True
   from pre have rule-case-dijs1: ∃ X. Γ,γ,p⊢ ⟨[Rule m a], Undecided⟩ ⇒ Decision
X ⟹ ?thesis
     apply −
     apply(rule disjI1)
     apply(elim exE conjE, rename-tac X)
     apply(simp)
     apply(rule-tac x=Decision X in exI)
     apply(rule-tac t=Decision X in seq-cons)
      apply(simp add: r; fail)
     apply(simp add: decision; fail)
     done

    from pre have rule-case-dijs2: Γ,γ,p⊢ ⟨[Rule m a], Undecided⟩ ⇒ Undecided
⟹ ?thesis
     apply −
     apply(rule disjI2)
     apply(rule-tac x=r#rs-called1 in exI)
     apply(rule-tac x=rs-called2 in exI)
     apply(rule-tac x=m' in exI)
     apply(simp add: r)
     apply(rule-tac t=Undecided in seq-cons)
      apply(simp; fail)
     apply(simp;fail)
     done

    show ?thesis
    proof(cases a)
    case Accept show ?thesis
      apply(rule rule-case-dijs1)
      apply(rule-tac x=FinalAllow in exI)
      using True pre Accept by(simp add: accept)
    next
    case Drop show ?thesis
      apply(rule rule-case-dijs1)
      apply(rule-tac x=FinalDeny in exI)
      using True Drop by(simp add: deny)
    next
    case Log show ?thesis
      apply(rule rule-case-dijs2)
      using Log True by(simp add: log)
    next
    case Reject show ?thesis
      apply(rule rule-case-dijs1)
      apply(rule-tac x=FinalDeny in exI)
```

```
      using Reject True by(simp add: reject)
    next
    case (Call x5)
      have ∃ t. Γ,γ,p⊢ ⟨[Rule m (Call x5)], Undecided⟩ ⇒ t by(rule case-call)
(simp add: r Call)
      with Call pre True show ?thesis
      apply(simp)
      apply(elim exE, rename-tac t-called)
      apply(case-tac t-called)
       apply(simp)
       apply(rule disjI2)
       apply(rule-tac x=r#rs-called1 in exI)
       apply(rule-tac x=rs-called2 in exI)
       apply(rule-tac x=m' in exI)
       apply(simp add: r)
       apply(rule-tac t=Undecided in seq-cons)
        apply(simp add: r; fail)
       apply(simp; fail)
      apply(rule disjI1)
      apply(rule-tac x=t-called in exI)
      apply(rule-tac t=t-called in seq-cons)
       apply(simp add: r; fail)
      apply(simp add: decision; fail)
      done
    next
    case Empty show ?thesis
      apply(rule rule-case-dijs2)
      using Empty True by(simp add: pre empty)
    next
    case Return show ?thesis
     apply(rule disjI2)
     apply(rule-tac x=[] in exI)
     apply(rule-tac x=rs-called1 @ Rule m' Return # rs-called2 in exI)
     apply(rule-tac x=m in exI)
     using Return True pre by(simp add: skip r)
    qed(simp-all add: a-not)
  qed

  from IH have **: a ≠ Return ⟶ (∃ t. Γ,γ,p⊢ ⟨[Rule m a], Undecided⟩ ⇒ t)
⟹ ?case
  proof(elim disjE, goal-cases)
  case 2
    from this obtain rs-called1 m' rs-called2 where
      a1: rs = rs-called1 @ [Rule m' Return] @ rs-called2 and
       a2: matches γ m' p and a3: Γ,γ,p⊢ ⟨rs-called1, Undecided⟩ ⇒ Undecided
by blast
    show ?case
      apply(rule *)
      using a1 a2 a3 by simp
```

```
  next
  case 1 thus ?case
   proof(cases a ≠ Return)
   case True
    with 1 obtain t1 t2 where t1: Γ,γ,p⊢ ⟨[Rule m a], Undecided⟩ ⇒ t1
                          and t2: Γ,γ,p⊢ ⟨rs, Undecided⟩ ⇒ t2 by blast
    from t1 t2 show ?thesis
    apply −
    apply(rule disjI1)
    apply(simp add: r)
    apply(cases t1)
     apply(simp-all)
     apply(rule-tac x=t2 in exI)
     apply(rule-tac seq'-cons)
      apply(simp-all)
    apply (meson decision seq-cons)
    done
   next
   case False show ?thesis
    proof(cases matches γ m p)
     assume ¬ matches γ m p with 1 show ?thesis
      apply −
      apply(rule disjI1)
      apply(elim exE)
      apply(rename-tac t')
      apply(rule-tac x=t' in exI)
      apply(rule-tac t=Undecided in seq-cons)
       apply(simp add: r nomatch; fail)
      by(simp)
    next
     assume matches γ m p with False show ?thesis
      apply −
      apply(rule disjI2)
      apply(rule-tac x=[] in exI)
      apply(rule-tac x=rs in exI)
      apply(rule-tac x=m in exI)
      apply(simp add: r skip; fail)
      done
    qed
   qed
  qed
  thus ?case using ex-neq-ret by blast
qed


lemma helper-defined-single:
  assumes wf (called-by-chain Γ)
  and ∀ rsg ∈ ran Γ ∪ {[Rule m a]}. wf-chain Γ rsg
   and ∀ rsg ∈ ran Γ ∪ {[Rule m a]}. ∀ r ∈ set rsg. (¬(∃ chain. get-action r =
```

```
Goto chain)) ∧ get-action r ≠ Unknown
    and a ≠ Return
    shows ∃ t. Γ,γ,p⊢ ⟨[Rule m a], s⟩ ⇒ t
  proof(cases s)
  case (Decision decision) thus ?thesis
    apply(rule-tac x=Decision decision in exI)
    apply(simp)
    using iptables-bigstep.decision by fast
  next
  case Undecided
    have ∃ t. Γ,γ,p⊢ ⟨[Rule m a], Undecided⟩ ⇒ t
    proof(cases matches γ m p)
    case False with assms show ?thesis
      apply(rule-tac x=Undecided in exI)
      apply(rule-tac t=Undecided in seq'-cons)
     apply (metis empty-iff empty-set insert-iff list.simps(15) nomatch' rule.sel(1))

      apply(simp add: skip; fail)
      done
    next
    case True
    show ?thesis
      proof(cases a)
      case Unknown with assms(3) show ?thesis by simp
      next
      case Goto with assms(3) show ?thesis by auto
      next
      case Accept with True show ?thesis by(auto intro: iptables-bigstep.intros)
      next
      case Drop with True show ?thesis by(auto intro: iptables-bigstep.intros)
      next
      case Reject with True show ?thesis by(auto intro: iptables-bigstep.intros)
      next
      case Log with True show ?thesis by(auto intro: iptables-bigstep.intros)
      next
      case Empty with True show ?thesis by(auto intro: iptables-bigstep.intros)
      next
      case Return with assms show ?thesis by simp
      next
      case (Call chain-name)
        thm wf-induct-rule[where r=(calls-chain Γ) and P=λx. ∃ t. Γ,γ,p⊢ ⟨[Rule
m (Call x)], Undecided⟩ ⇒ t]
        — Only the assumptions we will need
        from assms have wf (called-by-chain Γ)
            ∀ rsg∈ran Γ. wf-chain Γ rsg
          ∀ rsg∈ran Γ. ∀ r∈set rsg. (∀ chain. get-action r ≠ Goto chain) ∧ get-action
r ≠ Unknown by auto
        — strengthening the IH to do a well-founded induction
        hence matches γ m p ⟹ wf-chain Γ [Rule m (Call chain-name)] ⟹ (∃ t.
```

```
Γ,γ,p⊢ ⟨[Rule m (Call chain-name)], Undecided⟩ ⇒ t)
      proof(induction arbitrary: m rule: wf-induct-rule[where r=called-by-chain
Γ])
      case (less chain-name-neu)
     from less.prems have Γ chain-name-neu ≠ None by(simp add: wf-chain-def)
       from this obtain rs-called where rs-called: Γ chain-name-neu = Some
rs-called by blast

       from less rs-called have wf-chain Γ rs-called by (simp add: ranI)
       from less rs-called have rs-called ∈ ran Γ by (simp add: ranI)


       from less.prems rs-called have
        ∀ y m. ∀ r ∈ set rs-called. r = Rule m (Call y) ⟶ (y, chain-name-neu)
∈ called-by-chain Γ ∧ wf-chain Γ [Rule m (Call y)]
         apply(simp)
         apply(intro impI allI conjI)
          apply(simp add: called-by-chain-def)
          apply blast
         apply(simp add: wf-chain-def)
         apply (meson ranI rule.sel(2))
         done
        with less have ∀ y m. ∀ r∈set rs-called. r = Rule m (Call y) ⟶ (∃ t.
Γ,γ,p⊢ ⟨[Rule m (Call y)], Undecided⟩ ⇒ t)
         apply(intro allI, rename-tac y my)
         apply(case-tac matches γ my p)
          apply blast
         apply(intro ballI impI)
         apply(rule-tac x=Undecided in exI)
         apply(simp add: nomatch; fail)
         done
       from less.prems(4) rs-called ‹rs-called ∈ ran Γ›
      helper-cases-call-subchain-defined-or-return[OF less.prems(3) less.prems(4)
this ‹wf-chain Γ rs-called›] have
         (∃ t. Γ,γ,p⊢ ⟨rs-called, Undecided⟩ ⇒ t) ∨
          (∃ rs-called1 rs-called2 m'.
            Γ chain-name-neu = Some (rs-called1@[Rule m' Return]@rs-called2)
∧
              matches γ m' p ∧ Γ,γ,p⊢ ⟨rs-called1, Undecided⟩ ⇒ Undecided) by
simp
       thus ?case
       proof(elim disjE exE conjE)
         fix t
         assume a: Γ,γ,p⊢ ⟨rs-called, Undecided⟩ ⇒ t show ?case
         using call-result[OF less.prems(1) rs-called a] by(blast)
       next
         fix m' rs-called1 rs-called2
         assume a1: Γ chain-name-neu = Some (rs-called1 @ [Rule m' Return]
@ rs-called2)
```

and *a2*: *matches* γ m' p **and** *a3*: Γ,γ,p⊢ ⟨*rs-called1*, *Undecided*⟩ ⇒ *Undecided*
show *?case* **using** *call-return*[*OF less.prems*(*1*) *a1 a2 a3*] **by**(*blast*)
qed
qed
with *True assms Call* **show** *?thesis* **by** *simp*
qed
qed
with *Undecided* **show** *?thesis* **by** *simp*
qed

private lemma *helper-defined-ruleset-calledby*: *wf* (*called-by-chain* Γ) ⟹
∀ *rsg* ∈ *ran* Γ ∪ {*rs*}. *wf-chain* Γ *rsg* ⟹
∀ *rsg* ∈ *ran* Γ ∪ {*rs*}. ∀ *r* ∈ *set rsg*. (¬(∃ *chain*. *get-action r* = *Goto chain*)) ∧ *get-action r* ≠ *Unknown* ⟹
∀ *r* ∈ *set rs*. *get-action r* ≠ *Return* ⟹
∃ *t*. Γ,γ,p⊢ ⟨*rs*, *s*⟩ ⇒ *t*
apply(*rule iptables-bigstep-defined-if-singleton-rules*)
apply(*intro ballI*, *rename-tac r*, *case-tac r*, *rename-tac m a*, *simp*)
apply(*rule helper-defined-single*)
apply(*simp*; *fail*)
apply(*simp add*: *wf-chain-def*; *fail*)
apply *fastforce*
apply *fastforce*
done

corollary *semantics-bigstep-defined*: *finite* (*calls-chain* Γ) ⟹ *wf* (*calls-chain* Γ) ⟹ — call relation finite and terminating
∀ *rsg* ∈ *ran* Γ ∪ {*rs*}. *wf-chain* Γ *rsg* ⟹ — All calls to defined chains
∀ *rsg* ∈ *ran* Γ ∪ {*rs*}. ∀ *r* ∈ *set rsg*. (∀ *x*. *get-action r* ≠ *Goto x*) ∧ *get-action r* ≠ *Unknown* ⟹ — no bad actions
∀ *r* ∈ *set rs*. *get-action r* ≠ *Return* — no toplevel return ⟹
∃ *t*. Γ,γ,p⊢ ⟨*rs*, *s*⟩ ⇒ *t*
apply(*drule*(*1*) *wf-called-by-chain*)
apply(*thin-tac wf* (*calls-chain* Γ))
apply(*rule helper-defined-ruleset-calledby*)
apply(*simp-all*)
done
end

Common Algorithms

lemma *iptables-bigstep-rm-LogEmpty*: Γ,γ,p⊢ ⟨*rm-LogEmpty rs*, *s*⟩ ⇒ *t* ⟷ Γ,γ,p⊢ ⟨*rs*, *s*⟩ ⇒ *t*
proof(*induction rs arbitrary*: *s*)
case *Nil* **thus** *?case* **by**(*simp*)
next
case (*Cons r rs*)
have *step-IH*: (⋀*s*. Γ,γ,p⊢ ⟨*rs1*, *s*⟩ ⇒ *t* = Γ,γ,p⊢ ⟨*rs2*, *s*⟩ ⇒ *t*) ⟹

```
        Γ,γ,p⊢ ⟨r#rs1, s⟩ ⇒ t = Γ,γ,p⊢ ⟨r#rs2, s⟩ ⇒ t for rs1 rs2 r
  by (meson seq'-cons seqE-cons)
  have case-log: Γ,γ,p⊢ ⟨Rule m Log # rs, s⟩ ⇒ t ⟷ Γ,γ,p⊢ ⟨rs, s⟩ ⇒ t for m
   apply(rule iffI)
    apply(erule seqE-cons)
    apply (metis append-Nil log-remove seq')
   apply(rule-tac t=s in seq'-cons)
    apply(cases s)
     apply(cases matches γ m p)
      apply(simp add: log; fail)
     apply(simp add: nomatch; fail)
    apply(simp add: decision; fail)
   apply simp
  done
  have case-empty: Γ,γ,p⊢ ⟨Rule m Empty # rs, s⟩ ⇒ t ⟷ Γ,γ,p⊢ ⟨rs, s⟩ ⇒ t
for m
   apply(rule iffI)
    apply(erule seqE-cons)
    apply (metis append-Nil empty-empty seq')
   apply(rule-tac t=s in seq'-cons)
    apply(cases s)
     apply(cases matches γ m p)
      apply(simp add: empty; fail)
     apply(simp add: nomatch; fail)
    apply(simp add: decision; fail)
   apply simp
  done

  from Cons show ?case
  apply(cases r, rename-tac m a)
  apply(case-tac a)
        apply(simp-all)
        apply(simp-all cong: step-IH)
   apply(simp-all add: case-log case-empty)
  done
qed

lemma iptables-bigstep-rw-Reject: Γ,γ,p⊢ ⟨rw-Reject rs, s⟩ ⇒ t ⟷ Γ,γ,p⊢ ⟨rs, s⟩
⇒ t
proof(induction rs arbitrary: s)
case Nil thus ?case by(simp)
next
case (Cons r rs)
  have step-IH: (⋀s. Γ,γ,p⊢ ⟨rs1, s⟩ ⇒ t = Γ,γ,p⊢ ⟨rs2, s⟩ ⇒ t) ⟹
        Γ,γ,p⊢ ⟨r#rs1, s⟩ ⇒ t = Γ,γ,p⊢ ⟨r#rs2, s⟩ ⇒ t for rs1 rs2 r
  by (meson seq'-cons seqE-cons)
  have fst-rule: (⋀t. Γ,γ,p⊢ ⟨[r1], s⟩ ⇒ t ⟷ Γ,γ,p⊢ ⟨[r2], s⟩ ⇒ t) ⟹
    Γ,γ,p⊢ ⟨r1 # rs, s⟩ ⇒ t ⟷ Γ,γ,p⊢ ⟨r2 # rs, s⟩ ⇒ t for r1 r2 rs s t
  by (meson seq'-cons seqE-cons)
```

have *dropreject*: Γ,γ,*p*⊢ ⟨[*Rule m Drop*], *s*⟩ ⇒ *t* = Γ,γ,*p*⊢ ⟨[*Rule m Reject*], *s*⟩ ⇒ *t* **for** *m t*
apply(*cases s*)
apply(*cases matches* γ *m p*)
using *drop reject dropD rejectD* **apply** *fast*
using *nomatch nomatchD* **apply** *fast*
using *decision decisionD* **apply** *fast*
done

from *Cons* **show** *?case*
apply(*cases r*, *rename-tac m a*)
apply *simp*
apply(*case-tac a*)
apply(*simp-all*)
apply(*simp-all cong*: *step-IH*)
apply(*rule fst-rule*)
apply(*simp add*: *dropreject*)
done
qed

end
theory *Matching*
imports *Semantics*
begin

4.1 Boolean Matcher Algebra

lemma *MatchOr*: *matches* γ (*MatchOr m1 m2*) *p* ⟷ *matches* γ *m1 p* ∨ *matches* γ *m2 p*
by(*simp add*: *MatchOr-def*)

lemma *opt-MatchAny-match-expr-correct*: *matches* γ (*opt-MatchAny-match-expr m*) = *matches* γ *m*
proof −
have *matches* γ (*opt-MatchAny-match-expr-once m*) = *matches* γ *m* **for** *m*
apply(*simp add*: *fun-eq-iff*)
by(*induction m rule*: *opt-MatchAny-match-expr-once.induct*) (*simp-all*)
thus *?thesis*
apply(*simp add*: *opt-MatchAny-match-expr-def*)
apply(*rule repeat-stabilize-induct*)
by(*simp*)+
qed

lemma *matcheq-matchAny*: ¬ *has-primitive m* ⟹ *matcheq-matchAny m* ⟷ *matches* γ *m p*
by(*induction m*) *simp-all*

lemma *matcheq-matchNone*: ¬ *has-primitive m* ⟹ *matcheq-matchNone m* ⟷ ¬ *matches γ m p*
by(*auto dest*: *matcheq-matchAny matachAny-matchNone*)

lemma *matcheq-matchNone-not-matches*: *matcheq-matchNone m* ⟹ ¬ *matches γ m p*
by(*induction m rule*: *matcheq-matchNone.induct*) *auto*

Lemmas about matching in the *iptables-bigstep* semantics.

lemma *matches-rule-iptables-bigstep*:
assumes *matches γ m p* ⟷ *matches γ m′ p*
shows Γ,γ,*p*⊢ ⟨[*Rule m a*], *s*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨[*Rule m′ a*], *s*⟩ ⇒ *t* (**is** *?l* ⟷ *?r*)
proof −
{
fix *m m′*
assume Γ,γ,*p*⊢ ⟨[*Rule m a*], *s*⟩ ⇒ *t matches γ m p* ⟷ *matches γ m′ p*
hence Γ,γ,*p*⊢ ⟨[*Rule m′ a*], *s*⟩ ⇒ *t*
by (*induction* [*Rule m a*] *s t rule*: *iptables-bigstep-induct*)
(*auto intro*: *iptables-bigstep.intros simp*: *Cons-eq-append-conv dest*: *skipD*)
}
with *assms* **show** *?thesis* **by** *blast*
qed

lemma *matches-rule-and-simp-help*:
assumes *matches γ m p*
shows Γ,γ,*p*⊢ ⟨[*Rule* (*MatchAnd m m′*) *a′*], *Undecided*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨[*Rule m′ a′*], *Undecided*⟩ ⇒ *t* (**is** *?l* ⟷ *?r*)
proof
assume *?l* **thus** *?r*
by (*induction* [*Rule* (*MatchAnd m m′*) *a′*] *Undecided t rule*: *iptables-bigstep-induct*)
(*auto intro*: *iptables-bigstep.intros simp*: *assms Cons-eq-append-conv dest*: *skipD*)
next
assume *?r* **thus** *?l*
by (*induction* [*Rule m′ a′*] *Undecided t rule*: *iptables-bigstep-induct*)
(*auto intro*: *iptables-bigstep.intros simp*: *assms Cons-eq-append-conv dest*: *skipD*)
qed

lemma *matches-MatchNot-simp*:
assumes *matches γ m p*
shows Γ,γ,*p*⊢ ⟨[*Rule* (*MatchNot m*) *a*], *Undecided*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨[], *Undecided*⟩ ⇒ *t* (**is** *?l* ⟷ *?r*)
proof
assume *?l* **thus** *?r*
by (*induction* [*Rule* (*MatchNot m*) *a*] *Undecided t rule*: *iptables-bigstep-induct*)
(*auto intro*: *iptables-bigstep.intros simp*: *assms Cons-eq-append-conv dest*: *skipD*)

next
assume *?r*
hence $t = Undecided$
by (*metis skipD*)
with *assms* **show** *?l*
by (*fastforce intro*: *nomatch*)
qed

lemma *matches-MatchNotAnd-simp*:
assumes *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash \langle[Rule\ (MatchAnd\ (MatchNot\ m)\ m')\ a], Undecided\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle[], Undecided\rangle \Rightarrow t$ (**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *?l* **thus** *?r*
by (*induction* [*Rule* (*MatchAnd* (*MatchNot m*) *m'*) *a*] *Undecided t rule*: *iptables-bigstep-induct*)
(*auto intro*: *iptables-bigstep.intros simp add*: *assms Cons-eq-append-conv dest*: *skipD*)
next
assume *?r*
hence $t = Undecided$
by (*metis skipD*)
with *assms* **show** *?l*
by (*fastforce intro*: *nomatch*)
qed

lemma *matches-rule-and-simp*:
assumes *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash \langle[Rule\ (MatchAnd\ m\ m')\ a'], s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle[Rule\ m'\ a'], s\rangle \Rightarrow t$
proof (*cases s*)
case *Undecided*
with *assms* **show** *?thesis*
by (*simp add*: *matches-rule-and-simp-help*)
next
case *Decision*
thus *?thesis* **by** (*metis decision decisionD*)
qed

lemma *iptables-bigstep-MatchAnd-comm*:
$\Gamma,\gamma,p\vdash \langle[Rule\ (MatchAnd\ m1\ m2)\ a], s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle[Rule\ (MatchAnd\ m2\ m1)\ a], s\rangle \Rightarrow t$
proof −
{ fix *m1 m2*
have $\Gamma,\gamma,p\vdash \langle[Rule\ (MatchAnd\ m1\ m2)\ a], s\rangle \Rightarrow t \Longrightarrow \Gamma,\gamma,p\vdash \langle[Rule\ (MatchAnd\ m2\ m1)\ a], s\rangle \Rightarrow t$
proof (*induction* [*Rule* (*MatchAnd m1 m2*) *a*] *s t rule*: *iptables-bigstep-induct*)
case *Seq* **thus** *?case*
by (*metis Nil-is-append-conv append-Nil butlast-append butlast-snoc seq*)

qed (*auto intro*: *iptables-bigstep.intros*)
}
thus *?thesis* **by** *blast*
qed

4.2 Add match

definition *add-match* :: *'a match-expr* ⇒ *'a rule list* ⇒ *'a rule list* **where**
add-match m rs = *map* (λ*r*. *case r of Rule m' a'* ⇒ *Rule* (*MatchAnd m m'*) *a'*) *rs*

lemma *add-match-split*: *add-match m* (*rs1*@*rs2*) = *add-match m rs1* @ *add-match m rs2*
unfolding *add-match-def*
by (*fact map-append*)

lemma *add-match-split-fst*: *add-match m* (*Rule m' a'* # *rs*) = *Rule* (*MatchAnd m m'*) *a'* # *add-match m rs*
unfolding *add-match-def*
by *simp*

lemma *add-match-distrib*:
Γ,γ,*p*⊢ ⟨*add-match m1* (*add-match m2 rs*), *s*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨*add-match m2* (*add-match m1 rs*), *s*⟩ ⇒ *t*
proof −
{
fix *m1 m2*
have Γ,γ,*p*⊢ ⟨*add-match m1* (*add-match m2 rs*), *s*⟩ ⇒ *t* ⟹ Γ,γ,*p*⊢ ⟨*add-match m2* (*add-match m1 rs*), *s*⟩ ⇒ *t*
proof (*induction rs arbitrary*: *s*)
case *Nil* **thus** *?case* **by** (*simp add*: *add-match-def*)
next
case (*Cons r rs*)
from *Cons* **obtain** *m a* **where** *r*: *r* = *Rule m a* **by** (*cases r*) *simp*
with *Cons.prems* **obtain** *ti* **where** *1*: Γ,γ,*p*⊢ ⟨[*Rule* (*MatchAnd m1* (*MatchAnd m2 m*)) *a*], *s*⟩ ⇒ *ti* **and** *2*: Γ,γ,*p*⊢ ⟨*add-match m1* (*add-match m2 rs*), *ti*⟩ ⇒ *t*
apply(*simp add*: *add-match-split-fst*)
apply(*erule seqE-cons*)
by *simp*
from *1 r* **have** *base*: Γ,γ,*p*⊢ ⟨[*Rule* (*MatchAnd m2* (*MatchAnd m1 m*)) *a*], *s*⟩ ⇒ *ti*
by (*metis matches.simps*(*1*) *matches-rule-iptables-bigstep*)
from *2 Cons.IH* **have** *IH*: Γ,γ,*p*⊢ ⟨*add-match m2* (*add-match m1 rs*), *ti*⟩ ⇒ *t* **by** *simp*
from *base IH seq'-cons* **have** Γ,γ,*p*⊢ ⟨*Rule* (*MatchAnd m2* (*MatchAnd m1 m*)) *a* # *add-match m2* (*add-match m1 rs*), *s*⟩ ⇒ *t* **by** *fast*
thus *?case* **using** *r* **by**(*simp add*: *add-match-split-fst*[*symmetric*])

```
    qed
  }
  thus ?thesis by blast
qed

lemma add-match-split-fst': add-match m (a # rs) = add-match m [a] @ add-match
m rs
  by (simp add: add-match-split[symmetric])



lemma matches-add-match-simp:
  assumes m: matches γ m p
  shows Γ,γ,p⊢ ⟨add-match m rs, s⟩ ⇒ t ⟷ Γ,γ,p⊢ ⟨rs, s⟩ ⇒ t (is ?l ⟷ ?r)
  proof
    assume ?l with m show ?r
      proof (induction rs)
        case Nil
        thus ?case
          unfolding add-match-def by simp
      next
        case (Cons r rs)
       hence IH: Γ,γ,p⊢ ⟨add-match m rs, s⟩ ⇒ t ⟹ Γ,γ,p⊢ ⟨rs, s⟩ ⇒ t by(simp
add: add-match-split-fst)
        obtain m' a where r: r = Rule m' a by (cases r)
        with Cons.prems(2) obtain ti where Γ,γ,p⊢ ⟨[Rule (MatchAnd m m') a],
s⟩ ⇒ ti and Γ,γ,p⊢ ⟨add-match m rs, ti⟩ ⇒ t
          by(auto elim:seqE-cons simp add: add-match-split-fst)
        with Cons.prems(1) IH have Γ,γ,p⊢ ⟨[Rule m' a], s⟩ ⇒ ti by(simp add:
matches-rule-and-simp)
       with ‹Γ,γ,p⊢ ⟨add-match m rs, ti⟩ ⇒ t› IH r show ?case by(metis decision
state.exhaust iptables-bigstep-deterministic seq-cons)
      qed
  next
    assume ?r with m show ?l
      proof (induction rs)
        case Nil
        thus ?case
          unfolding add-match-def by simp
      next
        case (Cons r rs)
       hence IH: Γ,γ,p⊢ ⟨rs, s⟩ ⇒ t ⟹ Γ,γ,p⊢ ⟨add-match m rs, s⟩ ⇒ t by(simp
add: add-match-split-fst)
        obtain m' a where r: r = Rule m' a by (cases r)
        with Cons.prems(2) obtain ti where Γ,γ,p⊢ ⟨[Rule m' a], s⟩ ⇒ ti and
Γ,γ,p⊢ ⟨rs, ti⟩ ⇒ t
          by(auto elim:seqE-cons simp add: add-match-split-fst)
        with Cons.prems(1) IH have Γ,γ,p⊢ ⟨[Rule (MatchAnd m m') a], s⟩ ⇒ ti
by(simp add: matches-rule-and-simp)
```

```
      with ‹Γ,γ,p⊢ ⟨rs, ti⟩ ⇒ t› IH r show ?case
        apply(simp add: add-match-split-fst)
        by(metis decision state.exhaust iptables-bigstep-deterministic seq-cons)
    qed
  qed

lemma matches-add-match-MatchNot-simp:
  assumes m: matches γ m p
  shows Γ,γ,p⊢ ⟨add-match (MatchNot m) rs, s⟩ ⇒ t ⟷ Γ,γ,p⊢ ⟨[], s⟩ ⇒ t (is
?l s ⟷ ?r s)
  proof (cases s)
    case Undecided
    have ?l Undecided ⟷ ?r Undecided
      proof
        assume ?l Undecided with m show ?r Undecided
          proof (induction rs)
            case Nil
            thus ?case
              unfolding add-match-def by simp
          next
            case (Cons r rs)
            thus ?case
                  by (cases r) (metis matches-MatchNotAnd-simp skipD seqE-cons
add-match-split-fst)
          qed
      next
        assume ?r Undecided with m show ?l Undecided
          proof (induction rs)
            case Nil
            thus ?case
              unfolding add-match-def by simp
          next
            case (Cons r rs)
            thus ?case
                   by (cases r) (metis matches-MatchNotAnd-simp skipD seq'-cons
add-match-split-fst)
          qed
      qed
    with Undecided show ?thesis by fast
  next
    case (Decision d)
    thus ?thesis
      by(metis decision decisionD)
  qed

lemma not-matches-add-match-simp:
  assumes ¬ matches γ m p
  shows Γ,γ,p⊢ ⟨add-match m rs, Undecided⟩ ⇒ t ⟷ Γ,γ,p⊢ ⟨[], Undecided⟩ ⇒ t
  proof(induction rs)
```

case *Nil*
thus *?case*
unfolding *add-match-def* **by** *simp*
next
case (*Cons r rs*)
thus *?case*
by (*cases r*) (*metis assms add-match-split-fst matches.simps*(*1*) *nomatch seq'-cons nomatchD seqE-cons*)
qed

lemma *iptables-bigstep-add-match-notnot-simp*:
Γ,γ,*p*⊢ ⟨*add-match* (*MatchNot* (*MatchNot m*)) *rs*, *s*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨*add-match m rs*, *s*⟩ ⇒ *t*
proof(*induction rs*)
case *Nil*
thus *?case*
unfolding *add-match-def* **by** *simp*
next
case (*Cons r rs*)
thus *?case*
by (*cases r*)
(*metis decision decisionD state.exhaust matches.simps*(*2*) *matches-add-match-simp not-matches-add-match-simp*)
qed

lemma *add-match-match-not-cases*:
Γ,γ,*p*⊢ ⟨*add-match* (*MatchNot m*) *rs*, *Undecided*⟩ ⇒ *Undecided* ⟹ *matches* γ *m p* ∨ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided*
by (*metis matches.simps*(*2*) *matches-add-match-simp*)

lemma *not-matches-add-matchNot-simp*:
¬ *matches* γ *m p* ⟹ Γ,γ,*p*⊢ ⟨*add-match* (*MatchNot m*) *rs*, *s*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t*
by (*simp add*: *matches-add-match-simp*)

lemma *iptables-bigstep-add-match-and*:
Γ,γ,*p*⊢ ⟨*add-match m1* (*add-match m2 rs*), *s*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨*add-match* (*MatchAnd m1 m2*) *rs*, *s*⟩ ⇒ *t*
proof(*induction rs arbitrary*: *s t*)
case *Nil*
thus *?case*
unfolding *add-match-def* **by** *simp*
next
case(*Cons r rs*)
show *?case*
proof (*cases r*, *simp only*: *add-match-split-fst*)
fix *m a*

show Γ,γ,p⊢ ⟨*Rule* (*MatchAnd m1* (*MatchAnd m2 m*)) *a* # *add-match m1* (*add-match m2 rs*), *s*⟩ ⇒ *t* ⟷ Γ,γ,p⊢ ⟨*Rule* (*MatchAnd* (*MatchAnd m1 m2*) *m*) *a* # *add-match* (*MatchAnd m1 m2*) *rs*, *s*⟩ ⇒ *t* (**is** *?l* ⟷ *?r*)
proof
assume *?l* **with** *Cons.IH* **show** *?r*
apply −
apply(*erule seqE-cons*)
apply(*case-tac s*)
apply(*case-tac ti*)
apply (*metis matches.simps*(*1*) *matches-rule-and-simp matches-rule-and-simp-help nomatch seq′-cons*)
apply (*metis add-match-split-fst matches.simps*(*1*) *matches-add-match-simp not-matches-add-match-simp seq-cons*)
apply (*metis decision decisionD*)
done
next
assume *?r* **with** *Cons.IH* **show** *?l*
apply −
apply(*erule seqE-cons*)
apply(*case-tac s*)
apply(*case-tac ti*)
apply (*metis matches.simps*(*1*) *matches-rule-and-simp matches-rule-and-simp-help nomatch seq′-cons*)
apply (*metis add-match-split-fst matches.simps*(*1*) *matches-add-match-simp not-matches-add-match-simp seq-cons*)
apply (*metis decision decisionD*)
done
qed
qed
qed

lemma *optimize-matches-option-generic*:
assumes ∀ *r* ∈ *set rs*. *P* (*get-match r*)
and (⋀*m m′*. *P m* ⟹ *f m* = *Some m′* ⟹ *matches* γ *m′ p* = *matches* γ *m p*)
and (⋀*m*. *P m* ⟹ *f m* = *None* ⟹ ¬ *matches* γ *m p*)
shows Γ,γ,p⊢ ⟨*optimize-matches-option f rs*, *s*⟩ ⇒ *t* ⟷ Γ,γ,p⊢ ⟨*rs*, *s*⟩ ⇒ *t*
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *?rhs*
from *this assms* **show** *?lhs*
apply(*induction rs s t rule*: *iptables-bigstep-induct*)
apply(*auto simp*: *optimize-matches-option-append intro*: *iptables-bigstep.intros split*: *option.split*)
done
next
assume *?lhs*
from *this assms* **show** *?rhs*

```
    apply(induction f rs arbitrary: s rule: optimize-matches-option.induct)
     apply(simp; fail)
    apply(simp split: option.split-asm)
     apply(subgoal-tac ¬ matches γ m p)
     prefer 2 apply blast
    apply (metis decision nomatch seq'-cons state.exhaust)
    apply(erule seqE-cons)
    apply(rule-tac t=ti in seq'-cons)
     apply (meson matches-rule-iptables-bigstep)
    by blast
  qed

lemma optimize-matches-generic: ∀ r ∈ set rs. P (get-match r) ⟹
      (⋀m. P m ⟹ matches γ (f m) p = matches γ m p) ⟹
      Γ,γ,p⊢ ⟨optimize-matches f rs, s⟩ ⇒ t ⟷ Γ,γ,p⊢ ⟨rs, s⟩ ⇒ t
  unfolding optimize-matches-def
  apply(rule optimize-matches-option-generic)
    apply(simp; fail)
   apply(simp split: if-split-asm)
   apply blast
  apply(simp split: if-split-asm)
  using matcheq-matchNone-not-matches by fast
end
theory Ruleset-Update
imports Matching
begin

lemma free-return-not-match: Γ,γ,p⊢ ⟨[Rule m Return], Undecided⟩ ⇒ t ⟹ ¬
matches γ m p
  using no-free-return by fast
```

4.3 Background Ruleset Updating

```
lemma update-Gamma-nomatch:
  assumes ¬ matches γ m p
  shows Γ(chain ↦ Rule m a # rs),γ,p⊢ ⟨rs', s⟩ ⇒ t ⟷ Γ(chain ↦ rs),γ,p⊢
⟨rs', s⟩ ⇒ t (is ?l ⟷ ?r)
  proof
    assume ?l thus ?r
      proof (induction rs' s t rule: iptables-bigstep-induct)
        case (Call-return m a chain' rs₁ m' rs₂)
        thus ?case
          proof (cases chain' = chain)
            case True
            with Call-return show ?thesis
              apply simp
              apply(cases rs₁)
              using assms apply fastforce
              apply(rule-tac rs₁=list and m'=m' and rs₂=rs₂ in call-return)
```

```
            apply(simp)
           apply(simp)
          apply(simp)
         apply(simp)
          apply(erule seqE-cons[where Γ=(λa. if a = chain then Some rs else Γ a)])
          apply(frule iptables-bigstep-to-undecided[where Γ=(λa. if a = chain then Some rs else Γ a)])
         apply(simp)
         done
      qed (auto intro: call-return)
    next
      case (Call-result m' a' chain' rs' t')
      have Γ(chain ↦ rs),γ,p⊢ ⟨[Rule m' (Call chain')], Undecided⟩ ⇒ t'
        proof (cases chain' = chain)
          case True
           with Call-result have Rule m a # rs = rs' (Γ(chain ↦ rs)) chain' = Some rs
            by simp+
          with assms Call-result show ?thesis
            by (metis call-result nomatchD seqE-cons)
        next
          case False
          with Call-result show ?thesis
            by (metis call-result fun-upd-apply)
        qed
      with Call-result show ?case
        by fast
    qed (auto intro: iptables-bigstep.intros)
  next
    assume ?r thus ?l
    proof (induction rs' s t rule: iptables-bigstep-induct)
      case (Call-return m' a' chain' rs₁)
      thus ?case
        proof (cases chain' = chain)
          case True
          with Call-return show ?thesis
            using assms
            by (auto intro: seq-cons nomatch intro!: call-return[where rs₁ = Rule m a # rs₁])
        qed (auto intro: call-return)
    next
      case (Call-result m' a' chain' rs')
      thus ?case
        proof (cases chain' = chain)
          case True
          with Call-result show ?thesis
            using assms by (auto intro: seq-cons nomatch intro!: call-result)
        qed (auto intro: call-result)
```

```
    qed (auto intro: iptables-bigstep.intros)
  qed

lemma update-Gamma-log-empty:
  assumes a = Log ∨ a = Empty
  shows Γ(chain ↦ Rule m a # rs),γ,p⊢ ⟨rs', s⟩ ⇒ t ⟷
        Γ(chain ↦ rs),γ,p⊢ ⟨rs', s⟩ ⇒ t (is ?l ⟷ ?r)
  proof
    assume ?l thus ?r
      proof (induction rs' s t rule: iptables-bigstep-induct)
        case (Call-return m' a' chain' rs₁ m'' rs₂)

        note [simp] = fun-upd-apply[abs-def]

        from Call-return have Γ(chain ↦ rs),γ,p⊢ ⟨[Rule m' (Call chain')], Undecided⟩ ⇒ Undecided (is ?Call-return-case)
          proof(cases chain' = chain)
          case True with Call-return show ?Call-return-case
            — rs₁ cannot be empty
            proof(cases rs₁)
              case Nil with Call-return(3) ‹chain' = chain› assms have False by simp
              thus ?Call-return-case by simp
            next
            case (Cons r₁ rs₁s)
           from Cons Call-return have Γ(chain ↦ rs),γ,p⊢ ⟨r₁ # rs₁s, Undecided⟩ ⇒ Undecided by blast
            with seqE-cons[where Γ=Γ(chain ↦ rs)] obtain ti where
              Γ(chain ↦ rs),γ,p⊢ ⟨[r₁], Undecided⟩ ⇒ ti and Γ(chain ↦ rs),γ,p⊢ ⟨rs₁s, ti⟩ ⇒ Undecided by metis
          with iptables-bigstep-to-undecided[where Γ=Γ(chain ↦ rs)] have Γ(chain ↦ rs),γ,p⊢ ⟨rs₁s, Undecided⟩ ⇒ Undecided by fast
            with Cons Call-return ‹chain' = chain› show ?Call-return-case
              apply(rule-tac rs₁=rs₁s and m'=m'' and rs₂=rs₂ in call-return)
                apply(simp-all)
              done
            qed
          next
          case False with Call-return show ?Call-return-case
           by (auto intro: call-return)
          qed
        thus ?case using Call-return by blast
      next
        case (Call-result m' a' chain' rs' t')
        thus ?case
          proof (cases chain' = chain)
            case True
            with Call-result have rs' = [] @ [Rule m a] @ rs
              by simp
```

with *Call-result assms* **have** Γ(*chain* $\mapsto$ *rs*),γ,*p*$\vdash$ $\langle$[] @ *rs*, *Undecided*$\rangle$ $\Rightarrow$ *t'*
using *log-remove empty-empty* **by** *fast*
hence Γ(*chain* $\mapsto$ *rs*),γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *t'*
by *simp*
with *Call-result True* **show** *?thesis*
by (*metis call-result fun-upd-same*)
qed (*fastforce intro*: *call-result*)
qed (*auto intro*: *iptables-bigstep.intros*)
next
have *cases-a*: $\bigwedge P.\ (a = Log \Longrightarrow P\ a) \Longrightarrow (a = Empty \Longrightarrow P\ a) \Longrightarrow P\ a$ **using** *assms* **by** *blast*
assume *?r* **thus** *?l*
proof (*induction rs' s t rule*: *iptables-bigstep-induct*)
case (*Call-return m' a' chain' rs*$_1$ *m''* *rs*$_2$)
from *Call-return* **have** *xx*: Γ(*chain* $\mapsto$ *Rule m a # rs*),γ,*p*$\vdash$ $\langle$*Rule m a # rs*$_1$, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
apply −
apply(*rule cases-a*)
apply (*auto intro*: *nomatch seq-cons intro*!: *log empty simp del*: *fun-upd-apply*)
done
with *Call-return* **show** *?case*
proof(*cases chain' = chain*)
case *False*
with *Call-return* **have** *x*: (Γ(*chain* $\mapsto$ *Rule m a # rs*)) *chain' = Some* (*rs*$_1$ @ *Rule m'' Return # rs*$_2$)
by(*simp*)
with *Call-return* **have** Γ(*chain* $\mapsto$ *Rule m a # rs*),γ,*p*$\vdash$ $\langle$[*Rule m'* (*Call chain'*)], *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
apply −
apply(*rule call-return*[**where** *rs*$_1$=*rs*$_1$ **and** *m'*=*m''* **and** *rs*$_2$=*rs*$_2$])
apply(*simp-all add*: *x xx del*: *fun-upd-apply*)
done
thus Γ(*chain* $\mapsto$ *Rule m a # rs*),γ,*p*$\vdash$ $\langle$[*Rule m' a'*], *Undecided*$\rangle$ $\Rightarrow$ *Undecided* **using** *Call-return* **by** *simp*
next
case *True*
with *Call-return* **have** *x*: (Γ(*chain* $\mapsto$ *Rule m a # rs*)) *chain' = Some* (*Rule m a # rs*$_1$ @ *Rule m'' Return # rs*$_2$)
by(*simp*)
with *Call-return* **have** Γ(*chain* $\mapsto$ *Rule m a # rs*),γ,*p*$\vdash$ $\langle$[*Rule m'* (*Call chain'*)], *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
apply −
apply(*rule call-return*[**where** *rs*$_1$=*Rule m a*#*rs*$_1$ **and** *m'*=*m''* **and** *rs*$_2$=*rs*$_2$])
apply(*simp-all add*: *x xx del*: *fun-upd-apply*)
done
thus Γ(*chain* $\mapsto$ *Rule m a # rs*),γ,*p*$\vdash$ $\langle$[*Rule m' a'*], *Undecided*$\rangle$ $\Rightarrow$ *Undecided* **using** *Call-return* **by** *simp*

```
      qed
    next
      case (Call-result ma a chaina rs t)
      thus ?case
        apply (cases chaina = chain)
         apply(rule cases-a)
          apply (auto intro: nomatch seq-cons intro!: log empty call-result)[2]
        by (auto intro!: call-result)[1]
    qed (auto intro: iptables-bigstep.intros)
  qed

lemma map-update-chain-if: (λb. if b = chain then Some rs else Γ b) = Γ(chain
↦ rs)
  by auto

lemma no-recursive-calls-helper:
  assumes Γ,γ,p⊢ ⟨[Rule m (Call chain)], Undecided⟩ ⇒ t
  and     matches γ m p
  and     Γ chain = Some [Rule m (Call chain)]
  shows   False
  using assms
  proof (induction [Rule m (Call chain)] Undecided t rule: iptables-bigstep-induct)
    case Seq
    thus ?case
      by (metis Cons-eq-append-conv append-is-Nil-conv skipD)
  next
    case (Call-return chain′ rs1 m′ rs2)
    hence rs1 @ Rule m′ Return # rs2 = [Rule m (Call chain′)]
      by simp
    thus ?case
      by (cases rs1) auto
  next
    case Call-result
    thus ?case
      by simp
  qed (auto intro: iptables-bigstep.intros)

lemma no-recursive-calls:
  Γ(chain ↦ [Rule m (Call chain)]),γ,p⊢ ⟨[Rule m (Call chain)], Undecided⟩ ⇒ t
⟹ matches γ m p ⟹ False
  by (fastforce intro: no-recursive-calls-helper)

lemma no-recursive-calls2:
  assumes Γ(chain ↦ (Rule m (Call chain)) # rs′′),γ,p⊢ ⟨(Rule m (Call chain))
# rs′, Undecided⟩ ⇒ Undecided
  and     matches γ m p
  shows   False
  using assms
  proof (induction (Rule m (Call chain)) # rs′ Undecided Undecided arbitrary:
```

rs′ rule: *iptables-bigstep-induct*)
case (*Seq* rs_1 rs_2 *t*)
thus *?case*
by (*cases* rs_1) (*auto elim*: *seqE-cons simp add*: *iptables-bigstep-to-undecided*)
qed (*auto intro*: *iptables-bigstep.intros simp*: *Cons-eq-append-conv*)

lemma *update-Gamma-nochange1*:
assumes Γ(*chain* ↦ *rs*),γ,*p*⊢ ⟨[*Rule m a*], *Undecided*⟩ ⇒ *Undecided*
and Γ(*chain* ↦ *Rule m a* # *rs*),γ,*p*⊢ ⟨*rs′*, *s*⟩ ⇒ *t*
shows Γ(*chain* ↦ *rs*),γ,*p*⊢ ⟨*rs′*, *s*⟩ ⇒ *t*
using *assms*(*2*) **proof** (*induction rs′ s t rule*: *iptables-bigstep-induct*)
case (*Call-return m a chaina* rs_1 *m′* rs_2)
thus *?case*
proof (*cases chaina* = *chain*)
case *True*
with *Call-return* **show** *?thesis*
apply *simp*
apply(*cases* rs_1)
apply(*simp*)
using *assms* **apply** (*metis no-free-return*)
apply(*rule-tac* rs_1=*list* **and** *m′*=*m′* **and** rs_2=rs_2 **in** *call-return*)
apply(*simp*)
apply(*simp*)
apply(*simp*)
apply(*simp*)
apply(*erule seqE-cons*[**where** Γ=(λ*a*. *if a* = *chain then Some rs else* Γ *a*)])
apply(*frule iptables-bigstep-to-undecided*[**where** Γ=(λ*a*. *if a* = *chain then Some rs else* Γ *a*)])
apply(*simp*)
done
qed (*auto intro*: *call-return*)
next
case (*Call-result m a chaina rsa t*)
thus *?case*
proof (*cases chaina* = *chain*)
case *True*
with *Call-result* **show** *?thesis*
apply(*simp*)
apply(*cases rsa*)
apply(*simp*)
apply(*rule-tac rs*=*rs* **in** *call-result*)
apply(*simp-all*)
apply(*erule-tac seqE-cons*[**where** Γ=(λ*b*. *if b* = *chain then Some rs else* Γ *b*)])
apply(*case-tac t*)
apply(*simp*)
apply(*frule iptables-bigstep-to-undecided*[**where** Γ=(λ*b*. *if b* = *chain then*

Some rs else Γ *b*)])
apply(*simp*)
apply(*simp*)
apply(*subgoal-tac ti* = *Undecided*)
apply(*simp*)
using *assms*(*1*)[*simplified map-update-chain-if*[*symmetric*]] *iptables-bigstep-deterministic*
apply *fast*
done
qed (*fastforce intro*: *call-result*)
qed (*auto intro*: *iptables-bigstep.intros*)

lemma *update-gamme-remove-Undecidedpart*:
assumes Γ(*chain* ↦ *rs'*),γ,*p*⊢ ⟨*rs'*, *Undecided*⟩ ⇒ *Undecided*
and Γ(*chain* ↦ *rs1*@*rs'*),γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided*
shows Γ(*chain* ↦*rs'*),γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided*
using *assms*(*2*) **proof** (*induction rs Undecided Undecided rule*: *iptables-bigstep-induct*)
case *Seq*
thus *?case*
by (*auto simp*: *iptables-bigstep-to-undecided intro*: *seq*)
next
case (*Call-return m a chaina* rs_1 *m'* rs_2)
thus *?case*
apply(*cases chaina* = *chain*)
apply(*simp*)
apply(*cases length rs1* ≤ *length* rs_1)
apply(*simp add*: *List.append-eq-append-conv-if*)
apply(*rule-tac* rs_1=*drop* (*length rs1*) rs_1 **and** *m'*=*m'* **and** rs_2=rs_2 **in** *call-return*)
apply(*simp-all*)[*3*]
apply(*subgoal-tac* rs_1 = (*take* (*length rs1*) rs_1) @ *drop* (*length rs1*) rs_1)
prefer *2* **apply** (*metis append-take-drop-id*)
apply(*clarify*)
apply(*subgoal-tac* Γ(*chain* ↦ *drop* (*length rs1*) rs_1 @ *Rule m' Return* # rs_2),γ,*p*⊢
⟨(*take* (*length rs1*) rs_1) @ *drop* (*length rs1*) rs_1, *Undecided*⟩ ⇒ *Undecided*)
prefer *2* **apply**(*auto*)[*1*]
apply(*erule-tac* rs_1=*take* (*length rs1*) rs_1 **and** rs_2=*drop* (*length rs1*) rs_1 **in** *seqE*)
apply(*simp*)
apply(*frule-tac rs*=*drop* (*length rs1*) rs_1 **in** *iptables-bigstep-to-undecided*)
apply(*simp*; *fail*)
using *assms* **apply** (*auto intro*: *call-result call-return*)
done
next
case (*Call-result - - chain' rsa*)
thus *?case*
apply(*cases chain'* = *chain*)
apply(*simp*)
apply(*rule call-result*)

```
    apply(simp-all)[2]
   apply (metis iptables-bigstep-to-undecided seqE)
  apply (auto intro: call-result)
  done
qed (auto intro: iptables-bigstep.intros)

lemma update-Gamma-nocall:
 assumes ¬ (∃ chain. a = Call chain)
 shows Γ,γ,p⊢ ⟨[Rule m a], s⟩ ⇒ t ⟷ Γ',γ,p⊢ ⟨[Rule m a], s⟩ ⇒ t
 proof −
  {
   fix Γ Γ'
   have Γ,γ,p⊢ ⟨[Rule m a], s⟩ ⇒ t ⟹ Γ',γ,p⊢ ⟨[Rule m a], s⟩ ⇒ t
    proof (induction [Rule m a] s t rule: iptables-bigstep-induct)
     case Seq
      thus ?case by (metis (lifting, no-types) list-app-singletonE[where x =
Rule m a] skipD)
    next
     case Call-return thus ?case using assms by metis
    next
     case Call-result thus ?case using assms by metis
    qed (auto intro: iptables-bigstep.intros)
  }
  thus ?thesis
   by blast
 qed

lemma update-Gamma-call:
 assumes Γ chain = Some rs and Γ' chain = Some rs'
 assumes Γ,γ,p⊢ ⟨rs, Undecided⟩ ⇒ Undecided and Γ',γ,p⊢ ⟨rs', Undecided⟩ ⇒
Undecided
 shows Γ,γ,p⊢ ⟨[Rule m (Call chain)], s⟩ ⇒ t ⟷ Γ',γ,p⊢ ⟨[Rule m (Call chain)],
s⟩ ⇒ t
 proof −
  {
   fix Γ Γ' rs rs'
   assume assms:
    Γ chain = Some rs Γ' chain = Some rs'
    Γ,γ,p⊢ ⟨rs, Undecided⟩ ⇒ Undecided Γ',γ,p⊢ ⟨rs', Undecided⟩ ⇒ Undecided
   have Γ,γ,p⊢ ⟨[Rule m (Call chain)], s⟩ ⇒ t ⟹ Γ',γ,p⊢ ⟨[Rule m (Call chain)],
s⟩ ⇒ t
    proof (induction [Rule m (Call chain)] s t rule: iptables-bigstep-induct)
     case Seq
      thus ?case by (metis (lifting, no-types) list-app-singletonE[where x =
Rule m (Call chain)] skipD)
    next
     case Call-result
     thus ?case
      using assms by (metis call-result iptables-bigstep-deterministic)
```

```
      qed (auto intro: iptables-bigstep.intros assms)
    }
    note * = this
    show ?thesis
      using *[OF assms(1-4)] *[OF assms(2,1,4,3)] by blast
  qed

lemma update-Gamma-remove-call-undecided:
  assumes Γ(chain ↦ Rule m (Call foo) # rs'),γ,p⊢ ⟨rs, Undecided⟩ ⇒ Undecided
  and     matches γ m p
  shows Γ(chain ↦ rs'),γ,p⊢ ⟨rs, Undecided⟩ ⇒ Undecided
  using assms
  proof (induction rs Undecided Undecided arbitrary: rule: iptables-bigstep-induct)
    case Seq
    thus ?case
      by (force simp: iptables-bigstep-to-undecided intro: seq')
  next
    case (Call-return m a chaina rs₁ m' rs₂)
    thus ?case
      apply(cases chaina = chain)
      apply(cases rs₁)
      apply(force intro: call-return)
      apply(simp)
      apply(erule-tac Γ=Γ(chain ↦ list @ Rule m' Return # rs₂) in seqE-cons)
     apply(frule-tac Γ=Γ(chain ↦ list @ Rule m' Return # rs₂) in iptables-bigstep-to-undecided)
      apply(auto intro: call-return)
      done
  next
    case (Call-result m a chaina rsa)
    thus ?case
      apply(cases chaina = chain)
      apply(simp)
     apply (metis call-result fun-upd-same iptables-bigstep-to-undecided seqE-cons)
      apply (auto intro: call-result)
      done
  qed (auto intro: iptables-bigstep.intros)

lemma all-return-subchain:
  assumes a1: Γ chain = Some rs
  and     a2: matches γ m p
  and     a3: ∀ r∈set rs. get-action r = Return
  shows Γ,γ,p⊢ ⟨[Rule m (Call chain)], Undecided⟩ ⇒ Undecided
  proof (cases ∃ r ∈ set rs. matches γ (get-match r) p)
    case True
   hence (∃ rs1 r rs2. rs = rs1 @ r # rs2 ∧ matches γ (get-match r) p ∧ (∀ r'∈set
rs1. ¬ matches γ (get-match r') p))
      by (subst split-list-first-prop-iff[symmetric])
    then obtain rs1 r rs2
      where *: rs = rs1 @ r # rs2 matches γ (get-match r) p ∀ r'∈set rs1. ¬
```

matches γ (*get-match* r') p
by *auto*

with *a3* **obtain** m' **where** r = *Rule* m' *Return*
by (*cases* r) *simp*
with $*$ *assms* **show** *?thesis*
by (*fastforce intro*: *call-return nomatch'*)
next
case *False*
hence Γ,γ,$p\vdash$ $\langle rs,$ *Undecided*$\rangle \Rightarrow$ *Undecided*
by (*blast intro*: *nomatch'*)
with *a1 a2* **show** *?thesis*
by (*metis call-result*)
qed

lemma *get-action-case-simp*: *get-action* (*case* r *of Rule* m' $x \Rightarrow$ *Rule* (*MatchAnd* m m') x) = *get-action* r
by (*metis rule.case-eq-if rule.sel*(*2*))

lemma *updategamma-insert-new*: Γ,γ,$p\vdash$ $\langle rs,\ s\rangle \Rightarrow t \Longrightarrow$ *chain* $\notin$ *dom* $\Gamma \Longrightarrow$ Γ(*chain* $\mapsto$ *rs'*),γ,$p\vdash$ $\langle rs,\ s\rangle \Rightarrow t$
proof(*induction rule*: *iptables-bigstep-induct*)
case (*Call-result m a chain' rs t*)
thus *?case* **by** (*metis call-result domI fun-upd-def*)
next
case *Call-return*
thus *?case* **by** (*metis call-return domI fun-upd-def*)
qed(*auto intro*: *iptables-bigstep.intros*)

end
theory *Call-Return-Unfolding*
imports *Matching Ruleset-Update*
Common/Repeat-Stabilize
begin

5 *Call Return* Unfolding

Remove *Return*s

fun *process-ret* :: *'a rule list* $\Rightarrow$ *'a rule list* **where**
process-ret [] = [] |
process-ret (*Rule m Return* # *rs*) = *add-match* (*MatchNot m*) (*process-ret rs*) |
process-ret (*r*#*rs*) = *r* # *process-ret rs*

Remove *Call*s

fun *process-call* :: *'a ruleset* ⇒ *'a rule list* ⇒ *'a rule list* **where**
process-call Γ [] = [] |
process-call Γ (*Rule m* (*Call chain*) # *rs*) = *add-match m* (*process-ret* (*the* (Γ *chain*))) @ *process-call* Γ *rs* |
process-call Γ (*r*#*rs*) = *r* # *process-call* Γ *rs*

lemma *process-ret-split-fst-Return*:
a = *Return* ⟹ *process-ret* (*Rule m a* # *rs*) = *add-match* (*MatchNot m*) (*process-ret rs*)
by *auto*

lemma *process-ret-split-fst-NeqReturn*:
a ≠ *Return* ⟹ *process-ret*((*Rule m a*) # *rs*) = (*Rule m a*) # (*process-ret rs*)
by (*cases a*) *auto*

lemma *add-match-simp*: *add-match m* = *map* (λ*r*. *Rule* (*MatchAnd m* (*get-match r*)) (*get-action r*))
by (*auto simp*: *add-match-def cong*: *map-cong split*: *rule.split*)

definition *add-missing-ret-unfoldings* :: *'a rule list* ⇒ *'a rule list* ⇒ *'a rule list* **where**
add-missing-ret-unfoldings rs1 rs2 ≡
foldr (λ*rf acc*. *add-match* (*MatchNot* (*get-match rf*)) ∘ *acc*) [*r*←*rs1*. *get-action r* = *Return*] *id rs2*

fun *MatchAnd-foldr* :: *'a match-expr list* ⇒ *'a match-expr* **where**
MatchAnd-foldr [] = *undefined* |
MatchAnd-foldr [*e*] = *e* |
MatchAnd-foldr (*e* # *es*) = *MatchAnd e* (*MatchAnd-foldr es*)
fun *add-match-MatchAnd-foldr* :: *'a match-expr list* ⇒ (*'a rule list* ⇒ *'a rule list*) **where**
add-match-MatchAnd-foldr [] = *id* |
add-match-MatchAnd-foldr es = *add-match* (*MatchAnd-foldr es*)

lemma *add-match-add-match-MatchAnd-foldr*:
Γ,γ,*p*⊢ ⟨*add-match m* (*add-match-MatchAnd-foldr ms rs2*), *s*⟩ ⇒ *t* = Γ,γ,*p*⊢ ⟨*add-match* (*MatchAnd-foldr* (*m*#*ms*)) *rs2*, *s*⟩ ⇒ *t*
proof (*induction ms*)
case *Nil*
show *?case* **by** (*simp add*: *add-match-def*)
next
case *Cons*
thus *?case* **by** (*simp add*: *iptables-bigstep-add-match-and*)
qed

lemma *add-match-MatchAnd-foldr-empty-rs2*: *add-match-MatchAnd-foldr ms* [] = []

by (*induction ms*) (*simp-all add*: *add-match-def*)

lemma *add-missing-ret-unfoldings-alt*: Γ,γ,*p*⊢ ⟨*add-missing-ret-unfoldings rs1 rs2*, *s*⟩ ⇒ *t* ⟷
Γ,γ,*p*⊢ ⟨(*add-match-MatchAnd-foldr* (*map* (λ*r*. *MatchNot* (*get-match r*)) [*r*←*rs1*. *get-action r* = *Return*])) *rs2*, *s* ⟩ ⇒ *t*
proof(*induction rs1*)
case *Nil*
thus *?case*
unfolding *add-missing-ret-unfoldings-def* **by** *simp*
next
case (*Cons r rs*)
from *Cons* **obtain** *m a* **where** *r* = *Rule m a* **by**(*cases r*) (*simp*)
with *Cons* **show** *?case*
unfolding *add-missing-ret-unfoldings-def*
apply(*cases matches* γ *m p*)
apply (*simp-all add*: *matches-add-match-simp matches-add-match-MatchNot-simp add-match-add-match-MatchAnd-foldr*[*symmetric*])
done
qed

lemma *add-match-add-missing-ret-unfoldings-rot*:
Γ,γ,*p*⊢ ⟨*add-match m* (*add-missing-ret-unfoldings rs1 rs2*), *s*⟩ ⇒ *t* =
Γ,γ,*p*⊢ ⟨*add-missing-ret-unfoldings* (*Rule* (*MatchNot m*) *Return*#*rs1*) *rs2*, *s*⟩ ⇒ *t*
by(*simp add*: *add-missing-ret-unfoldings-def iptables-bigstep-add-match-notnot-simp*)

5.1 Completeness

lemma *process-ret-split-obvious*: *process-ret* (*rs*$_1$ @ *rs*$_2$) =
(*process-ret rs*$_1$) @ (*add-missing-ret-unfoldings rs*$_1$ (*process-ret rs*$_2$))
unfolding *add-missing-ret-unfoldings-def*
proof (*induction rs*$_1$ *arbitrary*: *rs*$_2$)
case (*Cons r rs*)
from *Cons* **obtain** *m a* **where** *r* = *Rule m a* **by** (*cases r*) *simp*
with *Cons.IH* **show** *?case*
apply(*cases a*)
apply(*simp-all add*: *add-match-split*)
done
qed *simp*

lemma *add-missing-ret-unfoldings-emptyrs2*: *add-missing-ret-unfoldings rs1* [] = []
unfolding *add-missing-ret-unfoldings-def*
by (*induction rs1*) (*simp-all add*: *add-match-def*)

lemma *process-call-split*: *process-call* Γ (*rs1* @ *rs2*) = *process-call* Γ *rs1* @ *process-call* Γ *rs2*
proof (*induction rs1*)

```
    case (Cons r rs1)
    thus ?case
      apply(cases r, rename-tac m a)
      apply(case-tac a)
            apply(simp-all)
      done
  qed simp

lemma process-call-split-fst: process-call Γ (a # rs) = process-call Γ [a] @ process-call Γ rs
  by (simp add: process-call-split[symmetric])


lemma iptables-bigstep-process-ret-undecided: Γ,γ,p⊢ ⟨rs, Undecided⟩ ⇒ t ⟹ Γ,γ,p⊢ ⟨process-ret rs, Undecided⟩ ⇒ t
proof (induction rs)
  case (Cons r rs)
  show ?case
    proof (cases r)
      case (Rule m' a')
      show Γ,γ,p⊢ ⟨process-ret (r # rs), Undecided⟩ ⇒ t
        proof (cases a')
          case Accept
          with Cons Rule show ?thesis
           by simp (metis acceptD decision decisionD nomatchD seqE-cons seq-cons)
        next
          case Drop
          with Cons Rule show ?thesis
            by simp (metis decision decisionD dropD nomatchD seqE-cons seq-cons)
        next
          case Log
          with Cons Rule show ?thesis
            by simp (metis logD nomatchD seqE-cons seq-cons)
        next
          case Reject
          with Cons Rule show ?thesis
           by simp (metis decision decisionD nomatchD rejectD seqE-cons seq-cons)
        next
          case (Call chain)
           from Cons.prems obtain ti where 1:Γ,γ,p⊢ ⟨[r], Undecided⟩ ⇒ ti and
2: Γ,γ,p⊢ ⟨rs, ti⟩ ⇒ t using seqE-cons by metis
          thus ?thesis
            proof(cases ti)
            case Undecided
              with Cons.IH 2 have IH: Γ,γ,p⊢ ⟨process-ret rs, Undecided⟩ ⇒ t by
simp
                from Undecided 1 Call Rule have Γ,γ,p⊢ ⟨[Rule m' (Call chain)],
Undecided⟩ ⇒ Undecided by simp
           with IH  have Γ,γ,p⊢ ⟨Rule m' (Call chain) # process-ret rs, Undecided⟩
```

⇒ *t* **using** *seq'-cons* **by** *fast*
thus *?thesis* **using** *Rule Call* **by** *force*
next
case (*Decision X*)
with *1 Rule Call* **have** Γ,γ,*p*⊢ ⟨[*Rule m'* (*Call chain*)], *Undecided*⟩ ⇒ *Decision X* **by** *simp*
moreover from *2 Decision* **have** *t* = *Decision X* **using** *decisionD* **by** *fast*
moreover from *decision* **have** Γ,γ,*p*⊢ ⟨*process-ret rs*, *Decision X*⟩ ⇒ *Decision X* **by** *fast*
ultimately show *?thesis* **using** *seq-cons* **by** (*metis Call Rule process-ret.simps*(*7*))
qed
next
case *Return*
with *Cons Rule* **show** *?thesis*
by *simp* (*metis matches.simps*(*2*) *matches-add-match-simp no-free-return nomatchD seqE-cons*)
next
case *Empty*
show *?thesis*
apply (*insert Empty Cons Rule*)
apply(*erule seqE-cons*)
apply (*rename-tac ti*)
apply(*case-tac ti*)
apply (*simp add*: *seq-cons*)
apply (*metis Rule-DecisionE emptyD state.distinct*(*1*))
done
next
case *Unknown*
show *?thesis* **using** *Unknown-actions-False*
by (*metis Cons.IH Cons.prems Rule Unknown nomatchD process-ret.simps*(*10*) *seqE-cons seq-cons*)
next
case *Goto* **thus** *?thesis* **using** *Unknown-actions-False*
by (*metis Cons.IH Cons.prems Rule Goto nomatchD process-ret.simps*(*8*) *seqE-cons seq-cons*)
qed
qed
qed *simp*

lemma *add-match-rot-add-missing-ret-unfoldings*:
Γ,γ,*p*⊢ ⟨*add-match m* (*add-missing-ret-unfoldings rs1 rs2*), *Undecided*⟩ ⇒ *Undecided* =
Γ,γ,*p*⊢ ⟨*add-missing-ret-unfoldings rs1* (*add-match m rs2*), *Undecided*⟩ ⇒ *Undecided*
apply(*simp add*: *add-missing-ret-unfoldings-alt add-match-add-missing-ret-unfoldings-rot add-match-add-match-MatchAnd-foldr*[*symmetric*] *iptables-bigstep-add-match-notnot-simp*)

apply(*cases map* (λ*r*. *MatchNot* (*get-match r*)) [*r*←*rs1* . (*get-action r*) = *Return*])
apply(*simp-all add*: *add-match-distrib*)
done

Completeness

theorem *unfolding-complete*: Γ,γ,*p*⊢ ⟨*rs*,*s*⟩ ⇒ *t* ⟹ Γ,γ,*p*⊢ ⟨*process-call* Γ *rs*,*s*⟩ ⇒ *t*
proof (*induction rule*: *iptables-bigstep-induct*)
case (*Nomatch m a*)
thus *?case*
by (*cases a*) (*auto intro*: *iptables-bigstep.intros simp add*: *not-matches-add-match-simp skip*)
next
case *Seq*
thus *?case*
by(*simp add*: *process-call-split seq'*)
next
case (*Call-return m a chain* rs_1 *m'* rs_2)
hence Γ,γ,*p*⊢ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided*
by *simp*
hence Γ,γ,*p*⊢ ⟨*process-ret* rs_1, *Undecided*⟩ ⇒ *Undecided*
by (*rule iptables-bigstep-process-ret-undecided*)
with *Call-return* **have** Γ,γ,*p*⊢ ⟨*process-ret* rs_1 @ *add-missing-ret-unfoldings* rs_1 (*add-match* (*MatchNot m'*) (*process-ret* rs_2)), *Undecided*⟩ ⇒ *Undecided*
by (*metis matches-add-match-MatchNot-simp skip add-match-rot-add-missing-ret-unfoldings seq'*)
with *Call-return* **show** *?case*
by (*simp add*: *matches-add-match-simp process-ret-split-obvious*)
next
case *Call-result*
thus *?case*
by (*simp add*: *matches-add-match-simp iptables-bigstep-process-ret-undecided*)
qed (*auto intro*: *iptables-bigstep.intros*)

lemma *process-ret-cases*:
process-ret rs = *rs* ∨ (∃ rs_1 rs_2 *m*. *rs* = rs_1@[*Rule m Return*]@rs_2 ∧ (*process-ret rs*) = rs_1@(*process-ret* ([*Rule m Return*]@rs_2)))
proof (*induction rs*)
case (*Cons r rs*)
thus *?case*
apply(*cases r*, *rename-tac m' a'*)
apply(*case-tac a'*)
apply(*simp-all*)
apply(*erule disjE*,*simp*,*rule disjI2*,*elim exE*,*simp add*: *process-ret-split-obvious*,
metis append-Cons process-ret-split-obvious process-ret.simps(*2*))+
apply(*rule disjI2*)
apply(*rule-tac x*=[] **in** *exI*)

apply(*rule-tac x=rs* **in** *exI*)
apply(*rule-tac x=m′* **in** *exI*)
apply(*simp*)
apply(*erule disjE,simp,rule disjI2,elim exE,simp add*: *process-ret-split-obvious*, *metis append-Cons process-ret-split-obvious process-ret.simps*(*2*))+
done
qed *simp*

lemma *process-ret-splitcases*:
obtains (*id*) *process-ret rs* = *rs*
| (*split*) rs_1 rs_2 *m* **where** *rs* = rs_1@[*Rule m Return*]@rs_2 **and** *process-ret rs* = rs_1@(*process-ret* ([*Rule m Return*]@rs_2))
by (*metis process-ret-cases*)

lemma *iptables-bigstep-process-ret-cases3*:
assumes Γ,γ,*p*⊢ ⟨*process-ret rs*, *Undecided*⟩ ⇒ *Undecided*
obtains (*noreturn*) Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided*
| (*return*) rs_1 rs_2 *m* **where** *rs* = rs_1@[*Rule m Return*]@rs_2 Γ,γ,*p*⊢ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided matches* γ *m p*
proof −
have Γ,γ,*p*⊢ ⟨*process-ret rs*, *Undecided*⟩ ⇒ *Undecided* ⟹
(Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided*) ∨
(∃ rs_1 rs_2 *m*. *rs* = rs_1@[*Rule m Return*]@rs_2 ∧ Γ,γ,*p*⊢ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* ∧ *matches* γ *m p*)
proof (*induction rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
from *Cons* **obtain** *m a* **where** *r*: *r* = *Rule m a* **by** (*cases r*) *simp*
from *r Cons* **show** *?case*
proof(*cases a* ≠ *Return*)
case *True*
with *r Cons.prems* **have** *prems-r*: Γ,γ,*p*⊢ ⟨[*Rule m a*], *Undecided*⟩ ⇒ *Undecided* **and** *prems-rs*: Γ,γ,*p*⊢ ⟨*process-ret rs*, *Undecided*⟩ ⇒ *Undecided*
apply(*simp-all add*: *process-ret-split-fst-NeqReturn*)
apply(*erule seqE-cons*, *frule iptables-bigstep-to-undecided*, *simp*)+
done
from *prems-rs Cons.IH* **have** Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided* ∨ (∃ rs_1 rs_2 *m*. *rs* = rs_1 @ [*Rule m Return*] @ rs_2 ∧ Γ,γ,*p*⊢ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* ∧ *matches* γ *m p*) **by** *simp*
thus Γ,γ,*p*⊢ ⟨*r* # *rs*, *Undecided*⟩ ⇒ *Undecided* ∨ (∃ rs_1 rs_2 *m*. *r* # *rs* = rs_1 @ [*Rule m Return*] @ rs_2 ∧ Γ,γ,*p*⊢ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* ∧ *matches* γ *m p*)
proof(*elim disjE*)
assume Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided*
hence Γ,γ,*p*⊢ ⟨*r* # *rs*, *Undecided*⟩ ⇒ *Undecided* **using** *prems-r* **by** (*metis r seq′-cons*)
thus *?thesis* **by** *simp*

next
assume ($\exists\, rs_1\ rs_2\ m.\ rs = rs_1\ @\ [Rule\ m\ Return]\ @\ rs_2 \wedge \Gamma,\gamma,p\vdash \langle rs_1, Undecided\rangle \Rightarrow Undecided \wedge matches\ \gamma\ m\ p$)
from *this* **obtain** $rs_1\ rs_2\ m'$ **where** $rs = rs_1\ @\ [Rule\ m'\ Return]\ @\ rs_2$ **and** $\Gamma,\gamma,p\vdash \langle rs_1, Undecided\rangle \Rightarrow Undecided$ **and** $matches\ \gamma\ m'\ p$ **by** *blast*
hence $\exists\, rs_1\ rs_2\ m.\ r\ \#\ rs = rs_1\ @\ [Rule\ m\ Return]\ @\ rs_2 \wedge \Gamma,\gamma,p\vdash \langle rs_1, Undecided\rangle \Rightarrow Undecided \wedge matches\ \gamma\ m\ p$
apply(*rule-tac x=Rule m a # rs*$_1$ **in** *exI*)
apply(*rule-tac x=rs*$_2$ **in** *exI*)
apply(*rule-tac x=m'* **in** *exI*)
apply(*simp add*: *r*)
using *prems-r seq'-cons* **by** *fast*
thus *?thesis* **by** *simp*
qed
next
case *False*
hence $a = Return$ **by** *simp*
with *Cons.prems r* **have** *prems*: $\Gamma,\gamma,p\vdash \langle add\text{-}match\ (MatchNot\ m)\ (process\text{-}ret\ rs), Undecided\rangle \Rightarrow Undecided$ **by** *simp*
show $\Gamma,\gamma,p\vdash \langle r\ \#\ rs, Undecided\rangle \Rightarrow Undecided \vee (\exists\, rs_1\ rs_2\ m.\ r\ \#\ rs = rs_1\ @\ [Rule\ m\ Return]\ @\ rs_2 \wedge \Gamma,\gamma,p\vdash \langle rs_1, Undecided\rangle \Rightarrow Undecided \wedge matches\ \gamma\ m\ p)$
proof(*cases matches* $\gamma\ m\ p$)
case *True*

hence $\exists\, rs_1\ rs_2\ m.\ r\ \#\ rs = rs_1\ @\ Rule\ m\ Return\ \#\ rs_2 \wedge \Gamma,\gamma,p\vdash \langle rs_1, Undecided\rangle \Rightarrow Undecided \wedge matches\ \gamma\ m\ p$
apply(*rule-tac x=[]* **in** *exI*)
apply(*rule-tac x=rs* **in** *exI*)
apply(*rule-tac x=m* **in** *exI*)
apply(*simp add*: *skip r* ‹*a = Return*›)
done
thus *?thesis* **by** *simp*
next
case *False*
with *nomatch seq-cons False r* **have** *r-nomatch*: $\bigwedge rs.\ \Gamma,\gamma,p\vdash \langle rs, Undecided\rangle \Rightarrow Undecided \Longrightarrow \Gamma,\gamma,p\vdash \langle r\ \#\ rs, Undecided\rangle \Rightarrow Undecided$ **by** *fast*
note *r-nomatch'=r-nomatch*[*simplified r* ‹*a = Return*›] — r unfolded
from *False not-matches-add-matchNot-simp prems* **have** $\Gamma,\gamma,p\vdash \langle process\text{-}ret\ rs, Undecided\rangle \Rightarrow Undecided$ **by** *fast*
with *Cons.IH* **have** *IH*: $\Gamma,\gamma,p\vdash \langle rs, Undecided\rangle \Rightarrow Undecided \vee (\exists\, rs_1\ rs_2\ m.\ rs = rs_1\ @\ [Rule\ m\ Return]\ @\ rs_2 \wedge \Gamma,\gamma,p\vdash \langle rs_1, Undecided\rangle \Rightarrow Undecided \wedge matches\ \gamma\ m\ p)$ **.**
thus *?thesis*
proof(*elim disjE*)
assume $\Gamma,\gamma,p\vdash \langle rs, Undecided\rangle \Rightarrow Undecided$
hence $\Gamma,\gamma,p\vdash \langle r\ \#\ rs, Undecided\rangle \Rightarrow Undecided$ **using** *r-nomatch* **by** *simp*
thus *?thesis* **by** *simp*

```
            next
             assume ∃ rs1 rs2 m. rs = rs1 @ [Rule m Return] @ rs2 ∧ Γ,γ,p⊢ ⟨rs1, Undecided⟩ ⇒ Undecided ∧ matches γ m p
              from this obtain rs1 rs2 m' where rs = rs1 @ [Rule m' Return] @ rs2 and Γ,γ,p⊢ ⟨rs1, Undecided⟩ ⇒ Undecided and matches γ m' p by blast
              hence ∃ rs1 rs2 m. r # rs = rs1 @ [Rule m Return] @ rs2 ∧ Γ,γ,p⊢ ⟨rs1, Undecided⟩ ⇒ Undecided ∧ matches γ m p
               apply(rule-tac x=Rule m Return # rs1 in exI)
               apply(rule-tac x=rs2 in exI)
               apply(rule-tac x=m' in exI)
               by(simp add: ‹a = Return› False r r-nomatch')
             thus ?thesis by simp
            qed
          qed
        qed
  qed
  with assms noreturn return show ?thesis by auto
qed

lemma iptables-bigstep-process-ret-DecisionD: Γ,γ,p⊢ ⟨process-ret rs, s⟩ ⇒ Decision X ⟹ Γ,γ,p⊢ ⟨rs, s⟩ ⇒ Decision X
proof (induction rs arbitrary: s)
  case (Cons r rs)
  thus ?case
    apply(cases r, rename-tac m a)
    apply(clarify)

    apply(case-tac a ≠ Return)
    apply(simp add: process-ret-split-fst-NeqReturn)
    apply(erule seqE-cons)
    apply(simp add: seq'-cons)

    apply(simp)

    apply(case-tac matches γ m p)
    apply(simp add: matches-add-match-MatchNot-simp skip)
    apply (metis decision skipD)


    apply(simp add: not-matches-add-matchNot-simp)
    by (metis decision state.exhaust nomatch seq'-cons)
qed simp
```

5.2 *process-ret* correctness

```
lemma process-ret-add-match-dist1: Γ,γ,p⊢ ⟨process-ret (add-match m rs), s⟩ ⇒ t ⟹ Γ,γ,p⊢ ⟨add-match m (process-ret rs), s⟩ ⇒ t
apply(induction rs arbitrary: s t)
 apply(simp add: add-match-def)
```

apply(*rename-tac r rs s t*)
apply(*case-tac r*)
apply(*rename-tac m′ a′*)
apply(*simp*)
apply(*case-tac a′*)
apply(*simp-all add*: *add-match-split-fst*)
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)
defer
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)
apply(*case-tac matches γ* (*MatchNot* (*MatchAnd m m′*)) *p*)
apply(*simp*)
apply (*meson seq′-cons seqE-cons*)
apply (*meson seq′-cons seqE-cons*)
by (*metis decision decisionD matches.simps*(*1*) *matches-add-match-MatchNot-simp matches-add-match-simp*
not-matches-add-matchNot-simp not-matches-add-match-simp state.exhaust)

lemma *process-ret-add-match-dist2*: Γ,γ,*p*⊢ ⟨*add-match m* (*process-ret rs*), *s*⟩ ⇒ *t* ⟹ Γ,γ,*p*⊢ ⟨*process-ret* (*add-match m rs*), *s*⟩ ⇒ *t*
apply(*induction rs arbitrary*: *s t*)
apply(*simp add*: *add-match-def*)
apply(*rename-tac r rs s t*)
apply(*case-tac r*)
apply(*rename-tac m′ a′*)
apply(*simp*)
apply(*case-tac a′*)
apply(*simp-all add*: *add-match-split-fst*)
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)

defer
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq′* **apply**(*fastforce*)
apply(*case-tac matches γ* (*MatchNot* (*MatchAnd m m′*)) *p*)
apply(*simp*)
apply (*meson seq′-cons seqE-cons*)
apply (*meson seq′-cons seqE-cons*)
by (*metis decision decisionD matches.simps*(*1*) *matches-add-match-MatchNot-simp matches-add-match-simp*
not-matches-add-matchNot-simp not-matches-add-match-simp state.exhaust)

lemma *process-ret-add-match-dist*: Γ,γ,*p*⊢ ⟨*process-ret* (*add-match m rs*), *s*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨*add-match m* (*process-ret rs*), *s*⟩ ⇒ *t*
by (*metis process-ret-add-match-dist1 process-ret-add-match-dist2*)

lemma *process-ret-Undecided-sound*:
assumes Γ(*chain* ↦ *rs*),γ,*p*⊢ ⟨*process-ret* (*add-match m rs*), *Undecided*⟩ ⇒ *Undecided*
shows Γ(*chain* ↦ *rs*),γ,*p*⊢ ⟨[*Rule m* (*Call chain*)], *Undecided*⟩ ⇒ *Undecided*
proof (*cases matches γ m p*)
case *False*
thus *?thesis*
by (*metis nomatch*)
next
case *True*
note *matches* = *this*
show *?thesis*
using *assms* **proof** (*induction rs*)
case *Nil*
from *call-result*[*OF matches*, **where** Γ=Γ(*chain* ↦ [])]
have (Γ(*chain* ↦ [])) *chain* = *Some* [] ⟹ Γ(*chain* ↦ []),γ,*p*⊢ ⟨[], *Undecided*⟩ ⇒ *Undecided* ⟹ Γ(*chain* ↦ []),γ,*p*⊢ ⟨[*Rule m* (*Call chain*)], *Undecided*⟩ ⇒ *Undecided*
by *simp*
thus *?case*
by (*fastforce intro*: *skip*)
next
case (*Cons r rs*)
obtain *m′ a′* **where** *r*: *r* = *Rule m′ a′* **by** (*cases r*) *blast*

with *Cons.prems* **have** *prems*: Γ(*chain* ↦ *Rule m′ a′* # *rs*),γ,*p*⊢ ⟨*process-ret* (*add-match m* (*Rule m′ a′* # *rs*)), *Undecided*⟩ ⇒ *Undecided*
by *fast*
hence *prems-simplified*: Γ(*chain* ↦ *Rule m′ a′* # *rs*),γ,*p*⊢ ⟨*process-ret* (*Rule m′ a′* # *rs*), *Undecided*⟩ ⇒ *Undecided*

using *matches* **by** (*metis matches-add-match-simp process-ret-add-match-dist*)

have Γ(*chain* ↦ *Rule m′ a′ # rs*),γ,*p*⊢ ⟨[*Rule m* (*Call chain*)], *Undecided*⟩ ⇒ *Undecided*
proof (*cases a′ = Return*)
case *True*
note *a′ = this*
have Γ(*chain* ↦ *Rule m′ Return # rs*),γ,*p*⊢ ⟨[*Rule m* (*Call chain*)], *Undecided*⟩ ⇒ *Undecided*
proof (*cases matches* γ *m′ p*)
case *True*
with *matches* **show** *?thesis*
by (*fastforce intro*: *call-return skip*)
next
case *False*
note *matches′ = this*
hence Γ(*chain* ↦ *rs*),γ,*p*⊢ ⟨*process-ret* (*Rule m′ a′ # rs*), *Undecided*⟩ ⇒ *Undecided*
by (*metis prems-simplified update-Gamma-nomatch*)
with *a′* **have** Γ(*chain* ↦ *rs*),γ,*p*⊢ ⟨*add-match* (*MatchNot m′*) (*process-ret rs*), *Undecided*⟩ ⇒ *Undecided*
by *simp*
with *matches matches′* **have** Γ(*chain* ↦ *rs*),γ,*p*⊢ ⟨*add-match m* (*process-ret rs*), *Undecided*⟩ ⇒ *Undecided*
by (*simp add*: *matches-add-match-simp not-matches-add-matchNot-simp*)
with *matches′ Cons.IH* **show** *?thesis*
by (*fastforce simp*: *update-Gamma-nomatch process-ret-add-match-dist*)
qed
with *a′* **show** *?thesis*
by *simp*
next
case *False*
note *a′ = this*
with *prems-simplified* **have** Γ(*chain* ↦ *Rule m′ a′ # rs*),γ,*p*⊢ ⟨*Rule m′ a′ # process-ret rs*, *Undecided*⟩ ⇒ *Undecided*
by (*simp add*: *process-ret-split-fst-NeqReturn*)
hence *step*: Γ(*chain* ↦ *Rule m′ a′ # rs*),γ,*p*⊢ ⟨[*Rule m′ a′*], *Undecided*⟩ ⇒ *Undecided*
and *IH-pre*: Γ(*chain* ↦ *Rule m′ a′ # rs*),γ,*p*⊢ ⟨*process-ret rs*, *Undecided*⟩ ⇒ *Undecided*
by (*metis seqE-cons iptables-bigstep-to-undecided*)+

from *step* **have** Γ(*chain* ↦ *rs*),γ,*p*⊢ ⟨*process-ret rs*, *Undecided*⟩ ⇒ *Undecided*
proof (*cases rule*: *Rule-UndecidedE*)
case *log* **thus** *?thesis*
using *IH-pre* **by** (*metis empty iptables-bigstep.log update-Gamma-nochange1 update-Gamma-nomatch*)
next

```
      case call thus ?thesis
        using IH-pre by (metis update-Gamma-remove-call-undecided)
      next
        case nomatch thus ?thesis
          using IH-pre by (metis update-Gamma-nomatch)
      qed

    hence Γ(chain ↦ rs),γ,p⊢ ⟨process-ret (add-match m rs), Undecided⟩ ⇒
Undecided
     by (metis matches matches-add-match-simp process-ret-add-match-dist)
      with Cons.IH have IH: Γ(chain ↦ rs),γ,p⊢ ⟨[Rule m (Call chain)],
Undecided⟩ ⇒ Undecided
      by fast

    from step show ?thesis
      proof (cases rule: Rule-UndecidedE)
        case log thus ?thesis using IH
          by (simp add: update-Gamma-log-empty)
      next
        case nomatch
        thus ?thesis
          using IH by (metis update-Gamma-nomatch)
      next
        case (call c)
        let ?Γ' = Γ(chain ↦ Rule m' a' # rs)
        from IH-pre show ?thesis
          proof (cases rule: iptables-bigstep-process-ret-cases3)
            case noreturn
             with call have ?Γ',γ,p⊢ ⟨Rule m' (Call c) # rs, Undecided⟩ ⇒
Undecided
              by (metis step seq-cons)
            from call have ?Γ' chain = Some (Rule m' (Call c) # rs)
              by simp
            from matches show ?thesis
              by (rule call-result) fact+
          next
            case (return rs₁ rs₂ new-m')
             with call have ?Γ' chain = Some ((Rule m' (Call c) # rs₁) @
[Rule new-m' Return] @ rs₂)
              by simp
               from call return step have ?Γ',γ,p⊢ ⟨Rule m' (Call c) # rs₁,
Undecided⟩ ⇒ Undecided
              using IH-pre by (auto intro: seq-cons)
            from matches show ?thesis
              by (rule call-return) fact+
          qed
      qed
  qed
thus ?case
```

```
      by (metis r)
    qed
  qed

lemma process-ret-Decision-sound:
  assumes Γ(chain ↦ rs),γ,p⊢ ⟨process-ret (add-match m rs), Undecided⟩ ⇒ Decision X
  shows Γ(chain ↦ rs),γ,p⊢ ⟨[Rule m (Call chain)], Undecided⟩ ⇒ Decision X
  proof (cases matches γ m p)
    case False
     thus ?thesis by (metis assms state.distinct(1) not-matches-add-match-simp process-ret-add-match-dist1 skipD)
  next
    case True
    note matches = this
    show ?thesis
      using assms proof (induction rs)
        case Nil
          hence False by (metis add-match-split append-self-conv state.distinct(1) process-ret.simps(1) skipD)
        thus ?case by simp
      next
        case (Cons r rs)
        obtain m' a' where r: r = Rule m' a' by (cases r) blast

       with Cons.prems have prems: Γ(chain ↦ Rule m' a' # rs),γ,p⊢ ⟨process-ret (add-match m (Rule m' a' # rs)), Undecided⟩ ⇒ Decision X
          by fast
       hence prems-simplified: Γ(chain ↦ Rule m' a' # rs),γ,p⊢ ⟨process-ret (Rule m' a' # rs), Undecided⟩ ⇒ Decision X
       using matches by (metis matches-add-match-simp process-ret-add-match-dist)

        have Γ(chain ↦ Rule m' a' # rs),γ,p⊢ ⟨[Rule m (Call chain)], Undecided⟩ ⇒ Decision X
          proof (cases a' = Return)
            case True
            note a' = this
              have Γ(chain ↦ Rule m' Return # rs),γ,p⊢ ⟨[Rule m (Call chain)], Undecided⟩ ⇒ Decision X
              proof (cases matches γ m' p)
                case True
                with matches prems-simplified a' show ?thesis
                  by (auto simp: not-matches-add-match-simp dest: skipD)
              next
                case False
                note matches' = this
                  with prems-simplified have Γ(chain ↦ rs),γ,p⊢ ⟨process-ret (Rule m' a' # rs), Undecided⟩ ⇒ Decision X
                  by (metis update-Gamma-nomatch)
```

```
                with a' matches matches' have Γ(chain ↦ rs),γ,p⊢ ⟨add-match m
(process-ret rs), Undecided⟩ ⇒ Decision X
              by (simp add: matches-add-match-simp not-matches-add-matchNot-simp)
                with matches matches' Cons.IH show ?thesis
              by (fastforce simp: update-Gamma-nomatch process-ret-add-match-dist
matches-add-match-simp not-matches-add-matchNot-simp)
            qed
           with a' show ?thesis
            by simp
         next
           case False
           with prems-simplified obtain ti
           where step: Γ(chain ↦ Rule m' a' # rs),γ,p⊢ ⟨[Rule m' a'], Undecided⟩
⇒ ti
                and IH-pre: Γ(chain ↦ Rule m' a' # rs),γ,p⊢ ⟨process-ret rs, ti⟩ ⇒
Decision X
            by (auto simp: process-ret-split-fst-NeqReturn elim: seqE-cons)

           hence Γ(chain ↦ Rule m' a' # rs),γ,p⊢ ⟨rs, ti⟩ ⇒ Decision X
            by (metis iptables-bigstep-process-ret-DecisionD)

           thus ?thesis
            using matches step by (force intro: call-result seq'-cons)
         qed
       thus ?case
        by (metis r)
     qed
  qed

lemma process-ret-result-empty: [] = process-ret rs ⟹ ∀ r ∈ set rs. get-action r
= Return
  proof (induction rs)
    case (Cons r rs)
    thus ?case
      apply(simp)
      apply(case-tac r)
      apply(rename-tac m a)
      apply(case-tac a)
            apply(simp-all add: add-match-def)
      done
  qed simp


lemma process-ret-sound':
  assumes Γ(chain ↦ rs),γ,p⊢ ⟨process-ret (add-match m rs), Undecided⟩ ⇒ t
  shows Γ(chain ↦ rs),γ,p⊢ ⟨[Rule m (Call chain)], Undecided⟩ ⇒ t
using assms by (metis state.exhaust process-ret-Undecided-sound process-ret-Decision-sound)
```

lemma *wf-chain-process-ret*: *wf-chain* Γ *rs* ⟹ *wf-chain* Γ (*process-ret rs*)
apply(*induction rs*)
apply(*simp add*: *wf-chain-def add-match-def*)
apply(*case-tac a*)
apply(*case-tac x2* ≠ *Return*)
apply(*simp add*: *process-ret-split-fst-NeqReturn*)
using *wf-chain-append* **apply** (*metis Cons-eq-appendI append-Nil*)
apply(*simp add*: *process-ret-split-fst-Return*)
apply(*simp add*: *wf-chain-def add-match-def get-action-case-simp*)
done
lemma *wf-chain-add-match*: *wf-chain* Γ *rs* ⟹ *wf-chain* Γ (*add-match m rs*)
by(*induction rs*) (*simp-all add*: *wf-chain-def add-match-def get-action-case-simp*)

5.3 Soundness

theorem *unfolding-sound*: *wf-chain* Γ *rs* ⟹ Γ,γ,*p*⊢ ⟨*process-call* Γ *rs*, *s*⟩ ⇒ *t* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t*
proof (*induction rs arbitrary*: *s t*)
case (*Cons r rs*)
thus *?case*
apply −
apply(*subst*(*asm*) *process-call-split-fst*)
apply(*erule seqE*)

unfolding *wf-chain-def*
apply(*case-tac r*, *rename-tac m a*)
apply(*case-tac a*)
apply(*simp-all add*: *seq′-cons*)

apply(*case-tac s*)
defer
apply (*metis decision decisionD*)
apply(*case-tac matches* γ *m p*)
defer
apply(*simp add*: *not-matches-add-match-simp*)
apply(*drule skipD*, *simp*)
apply (*metis nomatch seq-cons*)
apply(*clarify*)
apply(*simp add*: *matches-add-match-simp*)
apply(*rule-tac t*=*ti* **in** *seq-cons*)
apply(*simp-all*)

using *process-ret-sound′*
by (*metis fun-upd-triv matches-add-match-simp process-ret-add-match-dist*)
qed *simp*

corollary *unfolding-sound-complete*: *wf-chain* Γ *rs* ⟹ Γ,γ,*p*⊢ ⟨*process-call* Γ *rs*, *s*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t*

by (*metis unfolding-complete unfolding-sound*)

corollary *unfolding-n-sound-complete*: $\forall$ *rsg* $\in$ *ran* $\Gamma \cup \{rs\}$. *wf-chain* Γ *rsg* $\Longrightarrow$ Γ,γ,*p*$\vdash$ $\langle((\textit{process-call}\ \Gamma)\frown\frown n)\ rs,\ s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
proof(*induction n arbitrary*: *rs*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)
from *Suc* **have** $\Gamma,\gamma,p\vdash \langle(\textit{process-call}\ \Gamma\ \frown\frown\ n)\ rs,\ s\rangle \Rightarrow t = \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$ **by** *blast*
from *Suc.prems* **have** $\forall$ *a*$\in$*ran* $\Gamma \cup \{\textit{process-call}\ \Gamma\ rs\}$. *wf-chain* Γ *a*
proof(*induction rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case(*Cons r rs*)
from *Cons.prems* **have** $\forall$ *a*$\in$*ran* Γ. *wf-chain* Γ *a* **by** *blast*
from *Cons.prems* **have** *wf-chain* Γ [*r*]
apply(*simp*)
apply(*clarify*)
apply(*simp add*: *wf-chain-def*)
done
from *Cons.prems* **have** *wf-chain* Γ *rs*
apply(*simp*)
apply(*clarify*)
apply(*simp add*: *wf-chain-def*)
done
from *this Cons.prems Cons.IH* **have** *wf-chain* Γ (*process-call* Γ *rs*) **by** *blast*
from *this* ‹*wf-chain* Γ [*r*]›**have** *wf-chain* Γ (*r* # (*process-call* Γ *rs*)) **by**(*simp add*: *wf-chain-def*)
from *this Cons.prems* **have** *wf-chain* Γ (*process-call* Γ (*r*#*rs*))
apply(*cases r*)
apply(*rename-tac m a*, *clarify*)
apply(*case-tac a*)
apply(*simp-all*)
apply(*simp add*: *wf-chain-append*)
apply(*clarify*)
apply(*simp add*: ‹*wf-chain* Γ (*process-call* Γ *rs*)›)
apply(*rule wf-chain-add-match*)
apply(*rule wf-chain-process-ret*)
apply(*simp add*: *wf-chain-def*)
apply(*clarify*)
by (*metis ranI option.sel*)
from *this* ‹$\forall$ *a*$\in$*ran* Γ. *wf-chain* Γ *a*› **show** *?case* **by** *simp*
qed
from *this Suc.IH*[*of* ((*process-call* Γ) *rs*)] **have**
$\Gamma,\gamma,p\vdash \langle(\textit{process-call}\ \Gamma\ \frown\frown\ n)\ (\textit{process-call}\ \Gamma\ rs),\ s\rangle \Rightarrow t = \Gamma,\gamma,p\vdash \langle\textit{process-call}\ \Gamma\ rs,\ s\rangle \Rightarrow t$
by *simp*

from *this* **show** *?case* **by** (*simp add*: *Suc.prems funpow-swap1 unfolding-sound-complete*)
qed

```
loops in the linux kernel:
http://lxr.linux.no/linux+v3.2/net/ipv4/netfilter/ip_tables.c#L464
/* Figures out from what hook each rule can be called: returns 0 if
   there are loops.  Puts hook bitmask in comefrom. */
   static int mark_source_chains(const struct xt_table_info *newinfo,
                   unsigned int valid_hooks, void *entry0)

discussion: http://marc.info/?l=netfilter-devel&m=105190848425334&w=2
```

Example

lemma *process-call* [*''X''* ↦ [*Rule* (*Match b*) *Return*, *Rule* (*Match c*) *Accept*]] [*Rule* (*Match a*) (*Call ''X''*)] =
[*Rule* (*MatchAnd* (*Match a*) (*MatchAnd* (*MatchNot* (*Match b*)) (*Match c*))) *Accept*] **by** (*simp add*: *add-match-def*)

This is how a firewall processes a ruleset. It starts at a certain chain, usually INPUT, FORWARD, or OUTPUT (called *chain-name* in the lemma). The firewall has a default action of accept or drop. We can check *sanity-wf-ruleset* and the other assumptions at runtime. Consequently, we can apply *repeat-stabilize* as often as we want.

theorem *repeat-stabilize-process-call*:
assumes *sanity-wf-ruleset* Γ **and** *chain-name* ∈ *set* (*map fst* Γ) **and** *default-action* = *Accept* ∨ *default-action* = *Drop*
shows (*map-of* Γ),γ,*p*⊢ ⟨*repeat-stabilize n* (*process-call* (*map-of* Γ)) [*Rule MatchAny* (*Call chain-name*), *Rule MatchAny default-action*], *s*⟩ ⇒ *t* ⟷
(*map-of* Γ),γ,*p*⊢ ⟨[*Rule MatchAny* (*Call chain-name*), *Rule MatchAny default-action*], *s*⟩ ⇒ *t*
proof −
have *x*: *sanity-wf-ruleset* Γ ⟹ *rs* ∈ *ran* (*map-of* Γ) ⟹ *wf-chain* (*map-of* Γ) *rs* **for** Γ **and** *rs*::*'a rule list*
apply(*simp add*: *sanity-wf-ruleset-def wf-chain-def*)
by *fastforce*

from *assms*(*1*) **have** *1*: ∀ *rsg* ∈ *ran* (*map-of* Γ). *wf-chain* (*map-of* Γ) *rsg*
apply(*intro ballI*)
apply(*drule x*, *simp*)
apply(*simp*)
done
let *?rs*=[*Rule MatchAny* (*Call chain-name*), *Rule MatchAny default-action*]::*'a rule list*
from *assms*(*2*,*3*) **have** *2*: *wf-chain* (*map-of* Γ) *?rs*
apply(*simp add*: *wf-chain-def domD dom-map-of-conv-image-fst*)
by *blast*

have ∀ *rsg* ∈ *ran* Γ ∪ {*rs*}. *wf-chain* Γ *rsg* ⟹

Γ,γ,p⊢ ⟨*repeat-stabilize n* (*process-call* Γ) *rs*, *s*⟩ ⇒ *t* ⟷ Γ,γ,p⊢ ⟨*rs*, *s*⟩ ⇒ *t* **for** Γ *rs*
by(*simp add*: *repeat-stabilize-funpow unfolding-n-sound-complete*)
moreover from *1 2* **have** ∀ *rsg* ∈ *ran* (*map-of* Γ) ∪ {*?rs*}. *wf-chain* (*map-of* Γ) *rsg* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

definition *unfold-optimize-ruleset-CHAIN*
:: (*'a match-expr* ⇒ *'a match-expr*) ⇒ *string* ⇒ *action* ⇒ *'a ruleset* ⇒ *'a rule list option*
where
unfold-optimize-ruleset-CHAIN optimize chain-name default-action rs = (*let rs* =
(*repeat-stabilize 1000* (*optimize-matches opt-MatchAny-match-expr*)
(*optimize-matches optimize*
(*rw-Reject* (*rm-LogEmpty* (*repeat-stabilize 10000* (*process-call rs*)
[*Rule MatchAny* (*Call chain-name*), *Rule MatchAny default-action*]
)))))
in if simple-ruleset rs then Some rs else None)

lemma *unfold-optimize-ruleset-CHAIN*:
assumes *sanity-wf-ruleset* Γ **and** *chain-name* ∈ *set* (*map fst* Γ)
and *default-action* = *Accept* ∨ *default-action* = *Drop*
and ⋀*m*. *matches* γ (*optimize m*) *p* = *matches* γ *m p*
and *unfold-optimize-ruleset-CHAIN optimize chain-name default-action* (*map-of* Γ) = *Some rs*
shows (*map-of* Γ),γ,p⊢ ⟨*rs*, *s*⟩ ⇒ *t* ⟷
(*map-of* Γ),γ,p⊢ ⟨[*Rule MatchAny* (*Call chain-name*), *Rule MatchAny default-action*], *s*⟩ ⇒ *t*
proof −
from *assms*(*5*) **have** *rs*: *rs* = *repeat-stabilize 1000* (*optimize-matches opt-MatchAny-match-expr*)
(*optimize-matches optimize*
(*rw-Reject*
(*rm-LogEmpty*
(*repeat-stabilize 10000* (*process-call* (*map-of* Γ)) [*Rule MatchAny* (*Call chain-name*), *Rule MatchAny default-action*])))))
by(*simp add*: *unfold-optimize-ruleset-CHAIN-def Let-def split*: *if-split-asm*)

have *optimize-matches-generic-funpow-helper*: (⋀*m*. *matches* γ (*f m*) *p* = *matches* γ *m p*) ⟹
Γ,γ,p⊢ ⟨(*optimize-matches f* ^^ *n*) *rs*, *s*⟩ ⇒ *t* ⟷ Γ,γ,p⊢ ⟨*rs*, *s*⟩ ⇒ *t*
for Γ *f n rs*
proof(*induction n arbitrary*:)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*) **thus** *?case*

apply(*simp*)
apply(*subst optimize-matches-generic*[**where** *P*=λ-. *True*])
by *simp-all*
qed

have (*map-of* Γ),γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t* ⟷ *map-of* Γ,γ,*p*⊢ ⟨*repeat-stabilize 10000* (*process-call* (*map-of* Γ))
[*Rule MatchAny* (*Call chain-name*), *Rule MatchAny default-action*], *s*⟩ ⇒ *t*
apply(*simp add*: *rs repeat-stabilize-funpow*)
apply(*subst optimize-matches-generic-funpow-helper*)
apply (*simp add*: *opt-MatchAny-match-expr-correct*; *fail*)
apply(*subst optimize-matches-generic*[**where** *P*=λ-. *True*], *simp-all add*: *assms*(*4*))
apply(*simp add*: *iptables-bigstep-rw-Reject iptables-bigstep-rm-LogEmpty*)
done
also have ... ⟷ (*map-of* Γ),γ,*p*⊢ ⟨[*Rule MatchAny* (*Call chain-name*), *Rule MatchAny default-action*], *s*⟩ ⇒ *t*
using *assms*(*1*,*2*,*3*) **by**(*intro repeat-stabilize-process-call*[*of* Γ *chain-name default-action* γ *p 10000 s t*]) *simp-all*
finally show *?thesis* **.**
qed

end

6 Ternary Logic

theory *Ternary*
imports *Main*
begin

Kleene logic

datatype *ternaryvalue* = *TernaryTrue* | *TernaryFalse* | *TernaryUnknown*
datatype *ternaryformula* = *TernaryAnd ternaryformula ternaryformula*
| *TernaryOr ternaryformula ternaryformula*
| *TernaryNot ternaryformula*
| *TernaryValue ternaryvalue*

fun *ternary-to-bool* :: *ternaryvalue* ⇒ *bool option* **where**
ternary-to-bool TernaryTrue = *Some True* |
ternary-to-bool TernaryFalse = *Some False* |
ternary-to-bool TernaryUnknown = *None*

fun *bool-to-ternary* :: *bool* ⇒ *ternaryvalue* **where**
bool-to-ternary True = *TernaryTrue* |
bool-to-ternary False = *TernaryFalse*

lemma *the* ∘ *ternary-to-bool* ∘ *bool-to-ternary* = *id*
by(*simp add*: *fun-eq-iff*, *clarify*, *case-tac x*, *simp-all*)

lemma *ternary-to-bool-bool-to-ternary*: *ternary-to-bool* (*bool-to-ternary X*) = *Some X*
by(*cases X*, *simp-all*)
lemma *ternary-to-bool-None*: *ternary-to-bool t* = *None* ⟷ *t* = *TernaryUnknown*
by(*cases t*, *simp-all*)
lemma *ternary-to-bool-SomeE*: *ternary-to-bool t* = *Some X* ⟹
(*t* = *TernaryTrue* ⟹ *X* = *True* ⟹ *P*) ⟹ (*t* = *TernaryFalse* ⟹ *X* = *False* ⟹ *P*) ⟹ *P*
by(*cases t*)(*simp*)+
lemma *ternary-to-bool-Some*: *ternary-to-bool t* = *Some X* ⟷
(*t* = *TernaryTrue* ∧ *X* = *True*) ∨ (*t* = *TernaryFalse* ∧ *X* = *False*)
by(*cases t*, *simp-all*)
lemma *bool-to-ternary-Unknown*: *bool-to-ternary t* = *TernaryUnknown* ⟷ *False*
by(*cases t*, *simp-all*)

fun *eval-ternary-And* :: *ternaryvalue* ⇒ *ternaryvalue* ⇒ *ternaryvalue* **where**
eval-ternary-And TernaryTrue TernaryTrue = *TernaryTrue* |
eval-ternary-And TernaryTrue TernaryFalse = *TernaryFalse* |
eval-ternary-And TernaryFalse TernaryTrue = *TernaryFalse* |
eval-ternary-And TernaryFalse TernaryFalse = *TernaryFalse* |
eval-ternary-And TernaryFalse TernaryUnknown = *TernaryFalse* |
eval-ternary-And TernaryTrue TernaryUnknown = *TernaryUnknown* |
eval-ternary-And TernaryUnknown TernaryFalse = *TernaryFalse* |
eval-ternary-And TernaryUnknown TernaryTrue = *TernaryUnknown* |
eval-ternary-And TernaryUnknown TernaryUnknown = *TernaryUnknown*

lemma *eval-ternary-And-comm*: *eval-ternary-And t1 t2* = *eval-ternary-And t2 t1*
by (*cases t1 t2 rule*: *ternaryvalue.exhaust*[*case-product ternaryvalue.exhaust*]) *auto*

fun *eval-ternary-Or* :: *ternaryvalue* ⇒ *ternaryvalue* ⇒ *ternaryvalue* **where**
eval-ternary-Or TernaryTrue TernaryTrue = *TernaryTrue* |
eval-ternary-Or TernaryTrue TernaryFalse = *TernaryTrue* |
eval-ternary-Or TernaryFalse TernaryTrue = *TernaryTrue* |
eval-ternary-Or TernaryFalse TernaryFalse = *TernaryFalse* |
eval-ternary-Or TernaryTrue TernaryUnknown = *TernaryTrue* |
eval-ternary-Or TernaryFalse TernaryUnknown = *TernaryUnknown* |
eval-ternary-Or TernaryUnknown TernaryTrue = *TernaryTrue* |
eval-ternary-Or TernaryUnknown TernaryFalse = *TernaryUnknown* |
eval-ternary-Or TernaryUnknown TernaryUnknown = *TernaryUnknown*

fun *eval-ternary-Not* :: *ternaryvalue* ⇒ *ternaryvalue* **where**
eval-ternary-Not TernaryTrue = *TernaryFalse* |
eval-ternary-Not TernaryFalse = *TernaryTrue* |
eval-ternary-Not TernaryUnknown = *TernaryUnknown*

Just to hint that we did not make a typo, we add the truth table for the implication and show that it is compliant with $a \longrightarrow b = (\neg\ a \vee b)$

fun *eval-ternary-Imp* :: *ternaryvalue* ⇒ *ternaryvalue* ⇒ *ternaryvalue* **where**

eval-ternary-Imp TernaryTrue TernaryTrue = TernaryTrue |
eval-ternary-Imp TernaryTrue TernaryFalse = TernaryFalse |
eval-ternary-Imp TernaryFalse TernaryTrue = TernaryTrue |
eval-ternary-Imp TernaryFalse TernaryFalse = TernaryTrue |
eval-ternary-Imp TernaryTrue TernaryUnknown = TernaryUnknown |
eval-ternary-Imp TernaryFalse TernaryUnknown = TernaryTrue |
eval-ternary-Imp TernaryUnknown TernaryTrue = TernaryTrue |
eval-ternary-Imp TernaryUnknown TernaryFalse = TernaryUnknown |
eval-ternary-Imp TernaryUnknown TernaryUnknown = TernaryUnknown
lemma *eval-ternary-Imp a b = eval-ternary-Or (eval-ternary-Not a) b*
apply(*cases a*)
apply(*case-tac* [!] *b*)
apply(*simp-all*)
done

lemma *eval-ternary-Not-UnknownD*: *eval-ternary-Not t = TernaryUnknown ⟹ t = TernaryUnknown*
by (*cases t*) *auto*

lemma *eval-ternary-DeMorgan*:
eval-ternary-Not (eval-ternary-And a b) = eval-ternary-Or (eval-ternary-Not a) (eval-ternary-Not b)
eval-ternary-Not (eval-ternary-Or a b) = eval-ternary-And (eval-ternary-Not a) (eval-ternary-Not b)
by (*cases a b rule*: *ternaryvalue.exhaust*[*case-product ternaryvalue.exhaust*],*auto*)+

lemma *eval-ternary-idempotence-Not*: *eval-ternary-Not (eval-ternary-Not a) = a*
by (*cases a*) *simp-all*

lemma *eval-ternary-simps-simple*:
eval-ternary-And TernaryTrue x = x
eval-ternary-And x TernaryTrue = x
eval-ternary-And TernaryFalse x = TernaryFalse
eval-ternary-And x TernaryFalse = TernaryFalse
by(*case-tac* [!] *x*)(*simp-all*)

context
begin
private lemma *bool-to-ternary-simp1*: *bool-to-ternary X = TernaryTrue ⟷ X*
by (*metis bool-to-ternary.elims ternaryvalue.distinct*(*1*))
private lemma *bool-to-ternary-simp2*: *bool-to-ternary Y = TernaryFalse ⟷ ¬ Y*
by (*metis bool-to-ternary.elims ternaryvalue.distinct*(*1*))
private lemma *bool-to-ternary-simp3*: *eval-ternary-Not (bool-to-ternary X) = TernaryTrue ⟷ ¬ X*

by (*metis* (*full-types*) *bool-to-ternary-simp2 eval-ternary-Not.simps*(*1*) *eval-ternary-idempotence-Not*)
private lemma *bool-to-ternary-simp4*: *eval-ternary-Not* (*bool-to-ternary X*) = *TernaryFalse* ⟷ *X*
by (*metis bool-to-ternary-simp1 eval-ternary-Not.simps*(*1*) *eval-ternary-idempotence-Not*)
private lemma *bool-to-ternary-simp5*: ¬ (*eval-ternary-Not* (*bool-to-ternary X*) = *TernaryUnknown*)
by (*metis bool-to-ternary-Unknown eval-ternary-Not-UnknownD*)

private lemma *bool-to-ternary-simp6*: *bool-to-ternary X* ≠ *TernaryUnknown*
by (*metis* (*full-types*) *bool-to-ternary.simps*(*1*) *bool-to-ternary.simps*(*2*) *ternaryvalue.distinct*(*3*) *ternaryvalue.distinct*(*5*))

lemmas *bool-to-ternary-simps* = *bool-to-ternary-simp1 bool-to-ternary-simp2*
bool-to-ternary-simp3 bool-to-ternary-simp4
bool-to-ternary-simp5 bool-to-ternary-simp6
end

context
begin
private lemma *bool-to-ternary-pullup1*:
eval-ternary-Not (*bool-to-ternary X*) = *bool-to-ternary* (¬ *X*)
by(*cases X*)(*simp-all*)

private lemma *bool-to-ternary-pullup2*:
eval-ternary-And (*bool-to-ternary X1*) (*bool-to-ternary X2*) = *bool-to-ternary* (*X1* ∧ *X2*)
by (*metis bool-to-ternary-simps*(*1*) *bool-to-ternary-simps*(*2*) *eval-ternary-simps-simple*(*2*) *eval-ternary-simps-simple*(*4*))

private lemma *bool-to-ternary-pullup3*:
eval-ternary-Imp (*bool-to-ternary X1*) (*bool-to-ternary X2*) = *bool-to-ternary* (*X1* ⟶ *X2*)
by (*metis bool-to-ternary-simps*(*1*) *bool-to-ternary-simps*(*2*) *eval-ternary-Imp.simps*(*1*)
eval-ternary-Imp.simps(*2*) *eval-ternary-Imp.simps*(*3*) *eval-ternary-Imp.simps*(*4*))

private lemma *bool-to-ternary-pullup4*:
eval-ternary-Or (*bool-to-ternary X1*) (*bool-to-ternary X2*) = *bool-to-ternary* (*X1* ∨ *X2*)
by (*metis* (*full-types*) *bool-to-ternary.simps*(*1*) *bool-to-ternary.simps*(*2*) *eval-ternary-Or.simps*(*1*) *eval-ternary-Or.simps*(*2*) *eval-ternary-Or.simps*(*3*) *eval-ternary-Or.simps*(*4*))

lemmas *bool-to-ternary-pullup* = *bool-to-ternary-pullup1 bool-to-ternary-pullup2*
bool-to-ternary-pullup3 bool-to-ternary-pullup4
end

fun *ternary-ternary-eval* :: *ternaryformula* ⇒ *ternaryvalue* **where**

ternary-ternary-eval (*TernaryAnd t1 t2*) = *eval-ternary-And* (*ternary-ternary-eval t1*) (*ternary-ternary-eval t2*) |
ternary-ternary-eval (*TernaryOr t1 t2*) = *eval-ternary-Or* (*ternary-ternary-eval t1*) (*ternary-ternary-eval t2*) |
ternary-ternary-eval (*TernaryNot t*) = *eval-ternary-Not* (*ternary-ternary-eval t*) |
ternary-ternary-eval (*TernaryValue t*) = *t*

lemma *ternary-ternary-eval-DeMorgan*: *ternary-ternary-eval* (*TernaryNot* (*TernaryAnd a b*)) =
ternary-ternary-eval (*TernaryOr* (*TernaryNot a*) (*TernaryNot b*))
by (*simp add*: *eval-ternary-DeMorgan*)

lemma *ternary-ternary-eval-idempotence-Not*:
ternary-ternary-eval (*TernaryNot* (*TernaryNot a*)) = *ternary-ternary-eval a*
by (*simp add*: *eval-ternary-idempotence-Not*)

lemma *ternary-ternary-eval-TernaryAnd-comm*:
ternary-ternary-eval (*TernaryAnd t1 t2*) = *ternary-ternary-eval* (*TernaryAnd t2 t1*)
by (*simp add*: *eval-ternary-And-comm*)

lemma *eval-ternary-Not* (*ternary-ternary-eval t*) = (*ternary-ternary-eval* (*TernaryNot t*)) **by** *simp*

context
begin
private lemma *eval-ternary-simps-2*:
eval-ternary-And (*bool-to-ternary P*) *T* = *TernaryTrue* ⟷ *P* ∧ *T* = *TernaryTrue*
eval-ternary-And T (*bool-to-ternary P*) = *TernaryTrue* ⟷ *P* ∧ *T* = *TernaryTrue*
apply(*case-tac* [!] *P*)
apply(*simp-all add*: *eval-ternary-simps-simple*)
done

private lemma *eval-ternary-simps-3*:
eval-ternary-And (*ternary-ternary-eval x*) *T* = *TernaryTrue* ⟷
ternary-ternary-eval x = *TernaryTrue* ∧ *T* = *TernaryTrue*
eval-ternary-And T (*ternary-ternary-eval x*) = *TernaryTrue* ⟷
ternary-ternary-eval x = *TernaryTrue* ∧ *T* = *TernaryTrue*
apply(*case-tac* [!] *T*)
apply(*simp-all add*: *eval-ternary-simps-simple*)
apply(*case-tac* [!] (*ternary-ternary-eval x*))
apply(*simp-all*)
done

lemmas *eval-ternary-simps* = *eval-ternary-simps-simple eval-ternary-simps-2*

eval-ternary-simps-3
end

definition *ternary-eval* :: *ternaryformula* ⇒ *bool option* **where**
ternary-eval t = *ternary-to-bool* (*ternary-ternary-eval t*)

6.1 Negation Normal Form

A formula is in Negation Normal Form (NNF) if negations only occur at the atoms (not before and/or)

inductive *NegationNormalForm* :: *ternaryformula* ⇒ *bool* **where**
NegationNormalForm (*TernaryValue v*) |
NegationNormalForm (*TernaryNot* (*TernaryValue v*)) |
NegationNormalForm φ ⟹ *NegationNormalForm* ψ ⟹ *NegationNormalForm* (*TernaryAnd* φ ψ)|
NegationNormalForm φ ⟹ *NegationNormalForm* ψ ⟹ *NegationNormalForm* (*TernaryOr* φ ψ)

Convert a *ternaryformula* to a *ternaryformula* in NNF.

fun *NNF-ternary* :: *ternaryformula* ⇒ *ternaryformula* **where**
NNF-ternary (*TernaryValue v*) = *TernaryValue v* |
NNF-ternary (*TernaryAnd t1 t2*) = *TernaryAnd* (*NNF-ternary t1*) (*NNF-ternary t2*) |
NNF-ternary (*TernaryOr t1 t2*) = *TernaryOr* (*NNF-ternary t1*) (*NNF-ternary t2*) |
NNF-ternary (*TernaryNot* (*TernaryNot t*)) = *NNF-ternary t* |
NNF-ternary (*TernaryNot* (*TernaryValue v*)) = *TernaryValue* (*eval-ternary-Not v*) |
NNF-ternary (*TernaryNot* (*TernaryAnd t1 t2*)) = *TernaryOr* (*NNF-ternary* (*TernaryNot t1*)) (*NNF-ternary* (*TernaryNot t2*)) |
NNF-ternary (*TernaryNot* (*TernaryOr t1 t2*)) = *TernaryAnd* (*NNF-ternary* (*TernaryNot t1*)) (*NNF-ternary* (*TernaryNot t2*))

lemma *NNF-ternary-correct*: *ternary-ternary-eval* (*NNF-ternary t*) = *ternary-ternary-eval t*
proof(*induction t rule*: *NNF-ternary.induct*)
qed(*simp-all add*: *eval-ternary-DeMorgan eval-ternary-idempotence-Not*)

lemma *NNF-ternary-NegationNormalForm*: *NegationNormalForm* (*NNF-ternary t*)
proof(*induction t rule*: *NNF-ternary.induct*)
qed(*auto simp add*: *eval-ternary-DeMorgan eval-ternary-idempotence-Not intro*: *NegationNormalForm.intros*)

context
begin
private lemma *ternary-lift1*: *eval-ternary-Not tv* $\neq$ *TernaryFalse* $\longleftrightarrow$ *tv* = *TernaryFalse* $\vee$ *tv* = *TernaryUnknown*
using *eval-ternary-Not.elims* **by** *blast*
private lemma *ternary-lift2*: *eval-ternary-Not tv* $\neq$ *TernaryTrue* $\longleftrightarrow$ *tv* = *TernaryTrue* $\vee$ *tv* = *TernaryUnknown*
using *eval-ternary-Not.elims* **by** *blast*
private lemma *ternary-lift3*: *eval-ternary-Not tv* = *TernaryFalse* $\longleftrightarrow$ *tv* = *TernaryTrue*
by (*metis eval-ternary-Not.simps*(*1*) *eval-ternary-idempotence-Not*)
private lemma *ternary-lift4*: *eval-ternary-Not tv* = *TernaryTrue* $\longleftrightarrow$ *tv* = *TernaryFalse*
by (*metis eval-ternary-Not.simps*(*1*) *eval-ternary-idempotence-Not*)
private lemma *ternary-lift5*: *eval-ternary-Not tv* = *TernaryUnknown* $\longleftrightarrow$ *tv* = *TernaryUnknown*
by (*metis eval-ternary-Not.simps*(*3*) *eval-ternary-idempotence-Not*)

private lemma *ternary-lift6*: *eval-ternary-And t1 t2* = *TernaryFalse* $\longleftrightarrow$ *t1* = *TernaryFalse* $\vee$ *t2* = *TernaryFalse*
using *eval-ternary-And.elims* **by** *blast*
private lemma *ternary-lift7*: *eval-ternary-And t1 t2* = *TernaryTrue* $\longleftrightarrow$ *t1* = *TernaryTrue* $\wedge$ *t2* = *TernaryTrue*
using *eval-ternary-And.elims* **by** *blast*

lemmas *ternary-lift* = *ternary-lift1 ternary-lift2 ternary-lift3 ternary-lift4 ternary-lift5 ternary-lift6 ternary-lift7*
end

context
begin
private lemma *l1*: *eval-ternary-Not tv* = *TernaryTrue* $\Longrightarrow$ *tv* = *TernaryFalse*
by (*metis eval-ternary-Not.simps*(*1*) *eval-ternary-idempotence-Not*)
private lemma *l2*: *eval-ternary-And t1 t2* = *TernaryFalse* $\Longrightarrow$ *t1* = *TernaryFalse* $\vee$ *t2* = *TernaryFalse*
using *eval-ternary-And.elims* **by** *blast*

lemmas *eval-ternaryD* = *l1 l2*
end

end
theory *Matching-Ternary*
imports *../Common/Ternary ../Firewall-Common*
begin

7 Packet Matching in Ternary Logic

The matcher for a primitive match expression $'a$

type-synonym (*'a*, *'packet*) *exact-match-tac*=*'a* ⇒ *'packet* ⇒ *ternaryvalue*

If the matching is *TernaryUnknown*, it can be decided by the action whether this rule matches. E.g. in doubt, we allow packets

type-synonym *'packet unknown-match-tac*=*action* ⇒ *'packet* ⇒ *bool*

type-synonym (*'a*, *'packet*) *match-tac*=((*'a*, *'packet*) *exact-match-tac* × *'packet unknown-match-tac*)

For a given packet, map a firewall *'a match-expr* to a *ternaryformula* Evaluating the formula gives whether the packet/rule matches (or unknown).

fun *map-match-tac* :: (*'a*, *'packet*) *exact-match-tac* ⇒ *'packet* ⇒ *'a match-expr* ⇒ *ternaryformula* **where**
map-match-tac β p (*MatchAnd m1 m2*) = *TernaryAnd* (*map-match-tac β p m1*) (*map-match-tac β p m2*) |
map-match-tac β p (*MatchNot m*) = *TernaryNot* (*map-match-tac β p m*) |
map-match-tac β p (*Match m*) = *TernaryValue* (*β m p*) |
map-match-tac - - MatchAny = *TernaryValue TernaryTrue*

context
begin

the *ternaryformula*s we construct never have Or expressions.

private fun *ternary-has-or* :: *ternaryformula* ⇒ *bool* **where**
ternary-has-or (*TernaryOr - -*) ⟷ *True* |
ternary-has-or (*TernaryAnd t1 t2*) ⟷ *ternary-has-or t1* ∨ *ternary-has-or t2* |
ternary-has-or (*TernaryNot t*) ⟷ *ternary-has-or t* |
ternary-has-or (*TernaryValue -*) ⟷ *False*
private lemma *map-match-tac--does-not-use-TernaryOr*: ¬ (*ternary-has-or* (*map-match-tac β p m*))
by(*induction m*, *simp-all*)
declare *ternary-has-or.simps*[*simp del*]
end

fun *ternary-to-bool-unknown-match-tac* :: *'packet unknown-match-tac* ⇒ *action* ⇒ *'packet* ⇒ *ternaryvalue* ⇒ *bool* **where**
ternary-to-bool-unknown-match-tac - - - TernaryTrue = *True* |
ternary-to-bool-unknown-match-tac - - - TernaryFalse = *False* |
ternary-to-bool-unknown-match-tac α a p TernaryUnknown = *α a p*

Matching a packet and a rule:

1. Translate *'a match-expr* to ternary formula
2. Evaluate this formula

3. If *TernaryTrue*/ *TernaryFalse*, return this value
4. If *TernaryUnknown*, apply the *'a unknown-match-tac* to get a Boolean result

definition *matches* :: ('*a*, '*packet*) *match-tac* ⇒ '*a match-expr* ⇒ *action* ⇒ '*packet* ⇒ *bool* **where**
matches γ *m a p* ≡ *ternary-to-bool-unknown-match-tac* (*snd* γ) *a p* (*ternary-ternary-eval* (*map-match-tac* (*fst* γ) *p m*))

Alternative matches definitions, some more or less convenient

lemma *matches-tuple*: *matches* (β, α) *m a p* = *ternary-to-bool-unknown-match-tac* α *a p* (*ternary-ternary-eval* (*map-match-tac* β *p m*))
unfolding *matches-def* **by** *simp*

lemma *matches-case*: *matches* γ *m a p* ⟷ (*case ternary-eval* (*map-match-tac* (*fst* γ) *p m*) *of None* ⇒ (*snd* γ) *a p* | *Some b* ⇒ *b*)
unfolding *matches-def ternary-eval-def*
by (*cases* (*ternary-ternary-eval* (*map-match-tac* (*fst* γ) *p m*))) *auto*

lemma *matches-case-tuple*: *matches* (β, α) *m a p* ⟷ (*case ternary-eval* (*map-match-tac* β *p m*) *of None* ⇒ α *a p* | *Some b* ⇒ *b*)
by (*auto simp*: *matches-case split*: *option.splits*)

lemma *matches-case-ternaryvalue-tuple*: *matches* (β, α) *m a p* ⟷ (*case ternary-ternary-eval* (*map-match-tac* β *p m*) *of*
TernaryUnknown ⇒ α *a p* |
TernaryTrue ⇒ *True* |
TernaryFalse ⇒ *False*)
by(*simp split*: *option.split ternaryvalue.split add*: *matches-case ternary-to-bool-None ternary-eval-def*)

lemma *matches-casesE*:
matches (β, α) *m a p* ⟹
(*ternary-ternary-eval* (*map-match-tac* β *p m*) = *TernaryUnknown* ⟹ α *a p* ⟹ *P*) ⟹
(*ternary-ternary-eval* (*map-match-tac* β *p m*) = *TernaryTrue* ⟹ *P*)
⟹ *P*
proof(*induction m*)
qed(*auto split*: *option.split-asm simp*: *matches-case-tuple ternary-eval-def ternary-to-bool-bool-to-ternary elim*: *ternary-to-bool.elims*)

Example: ¬ *Unknown* is as good as *Unknown*

lemma ⟦ *ternary-ternary-eval* (*map-match-tac* β *p expr*) = *TernaryUnknown* ⟧ ⟹ *matches* (β, α) *expr a p* ⟷ *matches* (β, α) (*MatchNot expr*) *a p*
by(*simp add*: *matches-case-ternaryvalue-tuple*)

lemma *bunch-of-lemmata-about-matches*:
matches γ (*MatchAnd m1 m2*) *a p* $\longleftrightarrow$ *matches* γ *m1 a p* $\wedge$ *matches* γ *m2 a p*
matches γ *MatchAny a p*
matches γ (*MatchNot MatchAny*) *a p* $\longleftrightarrow$ *False*
matches γ (*MatchNot* (*MatchNot m*)) *a p* $\longleftrightarrow$ *matches* γ *m a p*
proof(*case-tac* [!] γ)
qed (*simp-all split*: *ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)

lemma *match-raw-bool*:
matches (β, α) (*Match expr*) *a p* = (*case ternary-to-bool* (β *expr p*) *of Some r* $\Rightarrow$ *r* | *None* $\Rightarrow$ (α *a p*))
by(*simp-all split*: *ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)
lemma *match-raw-ternary*:
matches (β, α) (*Match expr*) *a p* = (*case* (β *expr p*) *of TernaryTrue* $\Rightarrow$ *True* | *TernaryFalse* $\Rightarrow$ *False* | *TernaryUnknown* $\Rightarrow$ (α *a p*))
by(*simp-all split*: *ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)

lemma *matches-DeMorgan*: *matches* γ (*MatchNot* (*MatchAnd m1 m2*)) *a p* $\longleftrightarrow$ (*matches* γ (*MatchNot m1*) *a p*) $\vee$ (*matches* γ (*MatchNot m2*) *a p*)
by (*cases* γ) (*simp split*: *ternaryvalue.split add*: *matches-case-ternaryvalue-tuple eval-ternary-DeMorgan*)

7.1 Ternary Matcher Algebra

lemma *matches-and-comm*: *matches* γ (*MatchAnd m m'*) *a p* $\longleftrightarrow$ *matches* γ (*MatchAnd m' m*) *a p*
apply(*cases* γ, *rename-tac* β α, *clarify*)
by(*simp add*: *matches-case-ternaryvalue-tuple eval-ternary-And-comm*)

lemma *matches-not-idem*: *matches* γ (*MatchNot* (*MatchNot m*)) *a p* $\longleftrightarrow$ *matches* γ *m a p*
by (*fact bunch-of-lemmata-about-matches*)

lemma *MatchOr*: *matches* γ (*MatchOr m1 m2*) *a p* $\longleftrightarrow$ *matches* γ *m1 a p* $\vee$ *matches* γ *m2 a p*
by(*simp add*: *MatchOr-def matches-DeMorgan matches-not-idem*)

lemma *MatchOr-MatchNot*: *matches* γ (*MatchNot* (*MatchOr m1 m2*)) *a p* $\longleftrightarrow$ *matches* γ (*MatchNot m1*) *a p* $\wedge$ *matches* γ (*MatchNot m2*) *a p*
by(*simp add*: *MatchOr-def matches-DeMorgan bunch-of-lemmata-about-matches*)

lemma (*TernaryNot* (*map-match-tac* β *p* (*m*))) = (*map-match-tac* β *p* (*MatchNot m*))
by (*metis map-match-tac.simps*(*2*))

context
begin
private lemma *matches-simp1*: *matches* γ *m a p* $\Longrightarrow$ *matches* γ (*MatchAnd m m'*) *a p* $\longleftrightarrow$ *matches* γ *m' a p*
apply(*cases* γ, *rename-tac* β α, *clarify*)
apply(*simp split*: *ternaryvalue.split-asm ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)
done

private lemma *matches-simp11*: *matches* γ *m a p* $\Longrightarrow$ *matches* γ (*MatchAnd m' m*) *a p* $\longleftrightarrow$ *matches* γ *m' a p*
by(*simp-all add*: *matches-and-comm matches-simp1*)

private lemma *matches-simp2*: *matches* γ (*MatchAnd m m'*) *a p* $\Longrightarrow$ $\neg$ *matches* γ *m a p* $\Longrightarrow$ *False*
by (*simp add*: *bunch-of-lemmata-about-matches*)
private lemma *matches-simp22*: *matches* γ (*MatchAnd m m'*) *a p* $\Longrightarrow$ $\neg$ *matches* γ *m' a p* $\Longrightarrow$ *False*
by (*simp add*: *bunch-of-lemmata-about-matches*)

private lemma *matches-simp3*: *matches* γ (*MatchNot m*) *a p* $\Longrightarrow$ *matches* γ *m a p* $\Longrightarrow$ (*snd* γ) *a p*
apply(*cases* γ, *rename-tac* β α, *clarify*)
apply(*simp split*: *ternaryvalue.split-asm ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)
done
private lemma *matches* γ (*MatchNot m*) *a p* $\Longrightarrow$ *matches* γ *m a p* $\Longrightarrow$ (*ternary-eval* (*map-match-tac* (*fst* γ) *p m*)) = *None*
apply(*cases* γ, *rename-tac* β α, *clarify*)
apply(*simp split*: *ternaryvalue.split-asm ternaryvalue.split add*: *matches-case-ternaryvalue-tuple ternary-eval-def*)
done

lemmas *matches-simps* = *matches-simp1 matches-simp11*
lemmas *matches-dest* = *matches-simp2 matches-simp22*
end

lemma *matches-iff-apply-f-generic*: *ternary-ternary-eval* (*map-match-tac* β *p* (*f* (β,α) *a m*)) = *ternary-ternary-eval* (*map-match-tac* β *p m*) $\Longrightarrow$ *matches* (β,α) (*f* (β,α) *a m*) *a p* $\longleftrightarrow$ *matches* (β,α) *m a p*
by(*simp split*: *ternaryvalue.split-asm ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)

lemma *matches-iff-apply-f*: *ternary-ternary-eval* (*map-match-tac* β *p* (*f m*)) = *ternary-ternary-eval* (*map-match-tac* β *p m*) $\Longrightarrow$ *matches* (β,α) (*f m*) *a p* $\longleftrightarrow$ *matches* (β,α) *m a p*
by(*fact matches-iff-apply-f-generic*)

lemma *opt-MatchAny-match-expr-correct*: *matches* γ (*opt-MatchAny-match-expr* *m*) = *matches* γ *m*
proof(*case-tac* γ, *rename-tac* β α, *clarify*)
fix β α
assume $\gamma = (\beta, \alpha)$
have *ternary-ternary-eval* (*map-match-tac* β *p* (*opt-MatchAny-match-expr-once* *m*)) =
ternary-ternary-eval (*map-match-tac* β *p m*) **for** *p m*
proof(*induction m rule*: *opt-MatchAny-match-expr-once.induct*)
qed(*simp-all add*: *eval-ternary-simps eval-ternary-idempotence-Not*)
thus *matches* (β, α) (*opt-MatchAny-match-expr m*) = *matches* (β, α) *m*
apply(*simp add*: *fun-eq-iff*)
apply(*clarify*, *rename-tac a p*)
apply(*rule-tac f=opt-MatchAny-match-expr* **in** *matches-iff-apply-f*)
apply(*simp*)
apply(*simp add*: *opt-MatchAny-match-expr-def*)
apply(*rule repeat-stabilize-induct*)
by(*simp*)+
qed

An *'p unknown-match-tac* is wf if it behaves equal for *Reject* and *Drop*

definition *wf-unknown-match-tac* :: *'p unknown-match-tac* ⇒ *bool* **where**
wf-unknown-match-tac $\alpha \equiv (\alpha$ *Drop* = α *Reject*)

lemma *wf-unknown-match-tacD-False1*: *wf-unknown-match-tac* $\alpha \Longrightarrow \neg$ *matches* (β, α) *m Reject p* ⟹ *matches* (β, α) *m Drop p* ⟹ *False*
unfolding *wf-unknown-match-tac-def* **by**(*simp add*: *matches-case-ternaryvalue-tuple split*: *ternaryvalue.split-asm*)

lemma *wf-unknown-match-tacD-False2*: *wf-unknown-match-tac* $\alpha \Longrightarrow$ *matches* (β, α) *m Reject p* ⟹ ¬ *matches* (β, α) *m Drop p* ⟹ *False*
unfolding *wf-unknown-match-tac-def* **by**(*simp add*: *matches-case-ternaryvalue-tuple split*: *ternaryvalue.split-asm*)

7.2 Removing Unknown Primitives

definition *unknown-match-all* :: *'a unknown-match-tac* ⇒ *action* ⇒ *bool* **where**
unknown-match-all α *a* = (∀ *p*. α *a p*)
definition *unknown-not-match-any* :: *'a unknown-match-tac* ⇒ *action* ⇒ *bool* **where**
unknown-not-match-any α *a* = (∀ *p*. ¬ α *a p*)

fun *remove-unknowns-generic* :: (*'a*, *'packet*) *match-tac* ⇒ *action* ⇒ *'a match-expr* ⇒ *'a match-expr* **where**
remove-unknowns-generic - - MatchAny = *MatchAny* |
remove-unknowns-generic - - (*MatchNot MatchAny*) = *MatchNot MatchAny* |
remove-unknowns-generic (β, α) *a* (*Match A*) = (*if*

```
    (∀ p. ternary-ternary-eval (map-match-tac β p (Match A)) = TernaryUnknown)
    then
      if unknown-match-all α a then MatchAny else if unknown-not-match-any α a
then MatchNot MatchAny else Match A
    else (Match A)) |
  remove-unknowns-generic (β, α) a (MatchNot (Match A)) = (if
    (∀ p. ternary-ternary-eval (map-match-tac β p (Match A)) = TernaryUnknown)
    then
      if unknown-match-all α a then MatchAny else if unknown-not-match-any α a
then MatchNot MatchAny else MatchNot (Match A)
    else MatchNot (Match A)) |
  remove-unknowns-generic (β, α) a (MatchNot (MatchNot m)) = remove-unknowns-generic
(β, α) a m |
  remove-unknowns-generic (β, α) a (MatchAnd m1 m2) = MatchAnd
      (remove-unknowns-generic (β, α) a m1)
      (remove-unknowns-generic (β, α) a m2) |

  — ¬ (a ∧ b) = ¬ b ∨ ¬ a and ¬ Unknown = Unknown
  remove-unknowns-generic (β, α) a (MatchNot (MatchAnd m1 m2)) =
    (if (remove-unknowns-generic (β, α) a (MatchNot m1)) = MatchAny ∨
        (remove-unknowns-generic (β, α) a (MatchNot m2)) = MatchAny
        then MatchAny else
           (if (remove-unknowns-generic (β, α) a (MatchNot m1)) = MatchNot
MatchAny then
          remove-unknowns-generic (β, α) a (MatchNot m2) else
            if (remove-unknowns-generic (β, α) a (MatchNot m2)) = MatchNot
MatchAny then
          remove-unknowns-generic (β, α) a (MatchNot m1) else
            MatchNot (MatchAnd (MatchNot (remove-unknowns-generic (β, α) a
(MatchNot m1))) (MatchNot (remove-unknowns-generic (β, α) a (MatchNot m2)))))
      )

declare remove-unknowns-generic.simps(1−6) [code]

lemma [code]: remove-unknowns-generic γ a (MatchNot (MatchAnd m1 m2)) =
   (let m1′ = remove-unknowns-generic γ a (MatchNot m1); m2′ = remove-unknowns-generic
γ a (MatchNot m2) in
    (if m1′ = MatchAny ∨ m2′ = MatchAny
     then MatchAny
     else
       if m1′ = MatchNot MatchAny then m2′ else
       if m2′ = MatchNot MatchAny then m1′
     else
       MatchNot (MatchAnd (MatchNot m1′) (MatchNot m2′)))
      )
by(cases γ)(simp)


lemma remove-unknowns-generic-simp-3-4-unfolded: remove-unknowns-generic (β,
```

```
α) a (Match A) = (if
    (∀ p. ternary-ternary-eval (map-match-tac β p (Match A)) = TernaryUnknown)
    then
     if (∀ p. α a p) then MatchAny else if (∀ p. ¬ α a p) then MatchNot MatchAny
else Match A
    else (Match A))
 remove-unknowns-generic (β, α) a (MatchNot (Match A)) = (if
    (∀ p. ternary-ternary-eval (map-match-tac β p (Match A)) = TernaryUnknown)
    then
     if (∀ p. α a p) then MatchAny else if (∀ p. ¬ α a p) then MatchNot MatchAny
else MatchNot (Match A)
    else MatchNot (Match A))
  by(auto simp add: unknown-match-all-def unknown-not-match-any-def)

declare remove-unknowns-generic.simps[simp del]

lemmas remove-unknowns-generic-simps2 = remove-unknowns-generic.simps(1)
remove-unknowns-generic.simps(2)
           remove-unknowns-generic-simp-3-4-unfolded
            remove-unknowns-generic.simps(5) remove-unknowns-generic.simps(6)
remove-unknowns-generic.simps(7)


lemma matches (β, α) (remove-unknowns-generic (β, α) a (MatchNot (Match
A))) a p = matches (β, α) (MatchNot (Match A)) a p
by(simp add: remove-unknowns-generic-simps2 matches-case-ternaryvalue-tuple)



lemma remove-unknowns-generic: matches γ (remove-unknowns-generic γ a m) a
= matches γ m a
proof −
  have matches γ (remove-unknowns-generic γ a m) a p = matches γ m a p
  for p
  proof(induction γ a m rule: remove-unknowns-generic.induct)
  case 3 thus ?case
    by(simp add: bunch-of-lemmata-about-matches match-raw-ternary remove-unknowns-generic-simps2)
  next
  case 4 thus ?case
   by(simp add: matches-case-ternaryvalue-tuple remove-unknowns-generic-simps2)
  next
  case 7 thus ?case
   apply(simp add: bunch-of-lemmata-about-matches matches-DeMorgan remove-unknowns-generic-simps2)
    apply(simp add: matches-case-ternaryvalue-tuple)
    by fastforce
 qed(simp-all add: bunch-of-lemmata-about-matches remove-unknowns-generic-simps2)
  thus ?thesis by(simp add: fun-eq-iff)
qed
```

fun *has-unknowns* :: (*'a*, *'p*) *exact-match-tac* ⇒ *'a match-expr* ⇒ *bool* **where**
has-unknowns β (*Match A*) = (∃ *p*. *ternary-ternary-eval* (*map-match-tac β p* (*Match A*)) = *TernaryUnknown*) |
has-unknowns β (*MatchNot m*) = *has-unknowns β m* |
has-unknowns β MatchAny = *False* |
has-unknowns β (*MatchAnd m1 m2*) = (*has-unknowns β m1* ∨ *has-unknowns β m2*)

definition *packet-independent-α* :: *'p unknown-match-tac* ⇒ *bool* **where**
packet-independent-α α = (∀ *a p1 p2*. *a* = *Accept* ∨ *a* = *Drop* ⟶ *α a p1* ⟷ *α a p2*)

lemma *packet-independent-unknown-match*: *a* = *Accept* ∨ *a* = *Drop* ⟹ *packet-independent-α α* ⟹ ¬ *unknown-not-match-any α a* ⟷ *unknown-match-all α a*
by(*auto simp add*: *packet-independent-α-def unknown-match-all-def unknown-not-match-any-def*)

If for some type the exact matcher returns unknown, then it returns unknown for all these types

definition *packet-independent-β-unknown* :: (*'a*, *'packet*) *exact-match-tac* ⇒ *bool* **where**
packet-independent-β-unknown β ≡ ∀ *A*. (∃ *p*. *β A p* ≠ *TernaryUnknown*) ⟶ (∀ *p*. *β A p* ≠ *TernaryUnknown*)

lemma *remove-unknowns-generic-specification*: *a* = *Accept* ∨ *a* = *Drop* ⟹ *packet-independent-α α* ⟹
packet-independent-β-unknown β ⟹
¬ *has-unknowns β* (*remove-unknowns-generic* (*β*, *α*) *a m*)
proof(*induction* (*β*, *α*) *a m rule*: *remove-unknowns-generic.induct*)
case *3* **thus** *?case* **by**(*simp add*: *packet-independent-unknown-match packet-independent-β-unknown-def remove-unknowns-generic.simps*)
next
case *4* **thus** *?case* **by**(*simp add*: *packet-independent-unknown-match packet-independent-β-unknown-def remove-unknowns-generic.simps*)
qed(*simp-all add*: *remove-unknowns-generic.simps*)

Checking is something matches unconditionally

context
begin
private lemma *no-primitives-no-unknown*: ¬ *has-primitive m* ⟹ (*ternary-ternary-eval* (*map-match-tac β p m*)) ≠ *TernaryUnknown*
proof(*induction m*)
case *Match* **thus** *?case* **by** *auto*
next

case *MatchAny* **thus** *?case* **by** *simp*
next
case *MatchAnd* **thus** *?case* **by**(*auto elim*: *eval-ternary-And.elims*)
next
case *MatchNot* **thus** *?case* **by**(*auto dest*: *eval-ternary-Not-UnknownD*)
qed

private lemma *no-primitives-matchNot*: **assumes** ¬ *has-primitive m* **shows** *matches γ* (*MatchNot m*) *a p* ⟷ ¬ *matches γ m a p*
proof −
obtain *β α* **where** (*β*, *α*) = *γ* **by** (*cases γ*, *simp*)
thm *no-primitives-no-unknown*
from *assms* **have** *matches* (*β*, *α*) (*MatchNot m*) *a p* ⟷ ¬ *matches* (*β*, *α*) *m a p*
apply(*induction m*)
apply(*simp-all add*: *matches-case-ternaryvalue-tuple split*: *ternaryvalue.split*)
apply(*rename-tac m1 m2*)
by(*simp split*: *ternaryvalue.split-asm*)
with ‹(*β*, *α*) = *γ*› *assms* **show** *?thesis* **by** *simp*
qed

lemma *matcheq-matchAny*: ¬ *has-primitive m* ⟹ *matcheq-matchAny m* ⟷ *matches γ m a p*
proof(*induction m*)
case *Match* **hence** *False* **by** *auto*
thus *?case* **..**
next
case (*MatchNot m*)
from *MatchNot.prems* **have** ¬ *has-primitive m* **by** *simp*
with *no-primitives-matchNot* **have** *matches γ* (*MatchNot m*) *a p* = (¬ *matches γ m a p*) **by** *metis*
with *MatchNot* **show** *?case* **by**(*simp*)
next
case (*MatchAnd m1 m2*)
thus *?case* **by**(*simp add*: *bunch-of-lemmata-about-matches*)
next
case *MatchAny* **show** *?case* **by**(*simp add*: *Matching-Ternary.bunch-of-lemmata-about-matches*)
qed

lemma *matcheq-matchNone*: ¬ *has-primitive m* ⟹ *matcheq-matchNone m* ⟷ ¬ *matches γ m a p*
by(*auto dest*: *matcheq-matchAny matachAny-matchNone*)

lemma *matcheq-matchNone-not-matches*: *matcheq-matchNone m* ⟹ ¬ *matches γ m a p*
proof(*induction m rule*: *matcheq-matchNone.induct*)

qed(*auto simp add*: *bunch-of-lemmata-about-matches matches-DeMorgan*)

end

Lemmas about *MatchNot* in ternary logic.

lemma *matches-MatchNot-no-unknowns*:
  assumes ¬ *has-unknowns* β *m*
  shows *matches* (β,α) (*MatchNot m*) *a p* ⟷ ¬ *matches* (β,α) *m a p*
proof −
 { fix *m* **have** ¬ *has-unknowns* β *m* ⟹
     ternary-to-bool (*ternary-ternary-eval* (*map-match-tac* β *p m*)) ≠ *None*
   apply(*induction m*)
     apply(*simp-all*)
    using *ternary-to-bool.elims* **apply** *blast*
   using *ternary-to-bool-Some* **apply** *fastforce*
  using *ternary-lift*(*6*) *ternary-to-bool-Some* **by** *auto*
 } note *no-unknowns-ternary-to-bool-Some*=*this*
  from *assms* **show** *?thesis*
    by(*auto split*: *option.split-asm*
        simp: *matches-case-tuple no-unknowns-ternary-to-bool-Some ternary-to-bool-Some ternary-eval-def ternary-to-bool-bool-to-ternary*
          elim: *ternary-to-bool.elims*)
qed

lemma *MatchNot-ternary-ternary-eval*: (*ternary-ternary-eval* (*map-match-tac* β *p m*′)) = (*ternary-ternary-eval* (*map-match-tac* β *p m*)) ⟹
  matches (β,α) (*MatchNot m*′) *a p* = *matches* (β,α) (*MatchNot m*) *a p*
by(*simp add*: *matches-tuple*)

For our *′p unknown-match-tac*s *in-doubt-allow* and *in-doubt-deny*, when doing an induction over some function that modifies *m*, we get the *MatchNot* case for free (if we can set arbitrary *p*). This does not hold for arbitrary *′p unknown-match-tac*s.

lemma *matches-induction-case-MatchNot*:
    assumes α *Drop* ≠ α *Accept* **and** *packet-independent-α* α
    and ∀ *a*. *matches* (β,α) *m*′ *a p* = *matches* (β,α) *m a p*
    shows *matches* (β,α) (*MatchNot m*′) *a p* = *matches* (β,α) (*MatchNot m*) *a p*
proof −
 from *assms*(*1*) *assms*(*2*) **have** *xxxx-xxX*: ⋀*b*. ∀ *a*. α *a p* = (¬ *b*) ⟹ *False*
  apply(*simp add*: *packet-independent-α-def*)
  apply(*case-tac* α *Accept p*)
   apply(*simp-all*)
   apply(*case-tac* [!] α *Drop p*)
    apply(*simp-all add*: *fun-eq-iff*)
   apply *blast*+
  done

 have *xx2*: ⋀*t*. *ternary-eval* (*TernaryNot t*) = *None* ⟹ *ternary-eval t* = *None*

by (*simp add*: *eval-ternary-Not-UnknownD ternary-eval-def ternary-to-bool-None*)

have *xx3*: $\bigwedge t\ b.$ *ternary-eval* (*TernaryNot t*) = *Some b* $\Longrightarrow$ *ternary-eval t* = *Some* ($\neg$ *b*)
by (*metis eval-ternary-Not.simps*(*1*) *eval-ternary-Not.simps*(*2*) *ternary-eval-def ternary-ternary-eval.simps*(*3*) *ternary-ternary-eval-idempotence-Not ternary-to-bool-Some*)

from *assms* **show** *?thesis*
apply(*simp add*: *matches-case-tuple*)
apply(*case-tac ternary-eval* (*TernaryNot* (*map-match-tac* β *p m'*)))
apply(*case-tac* [!] *ternary-eval* (*TernaryNot* (*map-match-tac* β *p m*)))
apply(*simp-all*)
apply(*drule xx2*)
apply(*drule xx3*)
apply(*simp*)
using *xxxx-xxX* **apply** *metis*
apply(*drule xx2*)
apply(*drule xx3*)
apply(*simp*)
using *xxxx-xxX* **apply** *metis*
apply(*drule xx3*)+
apply(*simp*)
done
qed

end
theory *Semantics-Ternary*
imports *Matching-Ternary ../Common/List-Misc*
begin

8 Embedded Ternary-Matching Big Step Semantics

8.1 Ternary Semantics (Big Step)

inductive *approximating-bigstep* :: ('*a*, '*p*) *match-tac* $\Rightarrow$ '*p* $\Rightarrow$ '*a rule list* $\Rightarrow$ *state* $\Rightarrow$ *state* $\Rightarrow$ *bool*
($\langle$-,-$\vdash$ $\langle$-, -$\rangle$ $\Rightarrow_\alpha$ -$\rangle$ [*60*,*60*,*20*,*98*,*98*] *89*)
for γ **and** *p* **where**
skip: γ,*p*$\vdash$ $\langle$[], *t*$\rangle$ $\Rightarrow_\alpha$ *t* |
accept: $[\![$*matches* γ *m Accept p*$]\!]$ $\Longrightarrow$ γ,*p*$\vdash$ $\langle$[*Rule m Accept*], *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalAllow* |
drop: $[\![$*matches* γ *m Drop p*$]\!]$ $\Longrightarrow$ γ,*p*$\vdash$ $\langle$[*Rule m Drop*], *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalDeny* |
reject: $[\![$*matches* γ *m Reject p*$]\!]$ $\Longrightarrow$ γ,*p*$\vdash$ $\langle$[*Rule m Reject*], *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalDeny* |
log: $[\![$*matches* γ *m Log p*$]\!]$ $\Longrightarrow$ γ,*p*$\vdash$ $\langle$[*Rule m Log*], *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Undecided* |

empty: $[\![matches\ \gamma\ m\ Empty\ p]\!] \Longrightarrow \gamma,p\vdash \langle[Rule\ m\ Empty], Undecided\rangle \Rightarrow_\alpha Undecided$ |
nomatch: $[\![\neg\ matches\ \gamma\ m\ a\ p]\!] \Longrightarrow \gamma,p\vdash \langle[Rule\ m\ a], Undecided\rangle \Rightarrow_\alpha Undecided$ |
decision: $\gamma,p\vdash \langle rs, Decision\ X\rangle \Rightarrow_\alpha Decision\ X$ |
seq: $[\![\gamma,p\vdash \langle rs_1, Undecided\rangle \Rightarrow_\alpha t; \gamma,p\vdash \langle rs_2, t\rangle \Rightarrow_\alpha t']\!] \Longrightarrow \gamma,p\vdash \langle rs_1@rs_2, Undecided\rangle \Rightarrow_\alpha t'$

thm *approximating-bigstep.induct*[*of* γ p rs s t P]

lemma *approximating-bigstep-induct*[*case-names Skip Allow Deny Log Nomatch Decision Seq*, *induct pred*: *approximating-bigstep*] : $\gamma,p\vdash \langle rs,s\rangle \Rightarrow_\alpha t \Longrightarrow$
$(\bigwedge t.\ P\ []\ t\ t) \Longrightarrow$
$(\bigwedge m\ a.\ matches\ \gamma\ m\ a\ p \Longrightarrow a = Accept \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalAllow)) \Longrightarrow$
$(\bigwedge m\ a.\ matches\ \gamma\ m\ a\ p \Longrightarrow a = Drop \vee a = Reject \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalDeny)) \Longrightarrow$
$(\bigwedge m\ a.\ matches\ \gamma\ m\ a\ p \Longrightarrow a = Log \vee a = Empty \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided) \Longrightarrow$
$(\bigwedge m\ a.\ \neg\ matches\ \gamma\ m\ a\ p \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided) \Longrightarrow$
$(\bigwedge rs\ X.\ P\ rs\ (Decision\ X)\ (Decision\ X)) \Longrightarrow$
$(\bigwedge rs\ rs_1\ rs_2\ t\ t'.\ rs = rs_1\ @\ rs_2 \Longrightarrow \gamma,p\vdash \langle rs_1,Undecided\rangle \Rightarrow_\alpha t \Longrightarrow P\ rs_1\ Undecided\ t \Longrightarrow \gamma,p\vdash \langle rs_2,t\rangle \Rightarrow_\alpha t' \Longrightarrow P\ rs_2\ t\ t' \Longrightarrow P\ rs\ Undecided\ t')$
$\Longrightarrow P\ rs\ s\ t$
by (*induction rule*: *approximating-bigstep.induct*) (*simp-all*)

lemma *skipD*: $\gamma,p\vdash \langle[], s\rangle \Rightarrow_\alpha t \Longrightarrow s = t$
by (*induction* []::*'a rule list s t rule*: *approximating-bigstep-induct*) (*simp-all*)

lemma *decisionD*: $\gamma,p\vdash \langle rs, Decision\ X\rangle \Rightarrow_\alpha t \Longrightarrow t = Decision\ X$
by (*induction rs Decision X t rule*: *approximating-bigstep-induct*) (*simp-all*)

lemma *acceptD*: $\gamma,p\vdash \langle[Rule\ m\ Accept], Undecided\rangle \Rightarrow_\alpha t \Longrightarrow matches\ \gamma\ m\ Accept\ p \Longrightarrow t = Decision\ FinalAllow$
proof (*induction* [*Rule m Accept*] *Undecided t rule*: *approximating-bigstep-induct*)
case *Seq* **thus** *?case* **by** (*metis list-app-singletonE skipD*)
qed(*simp-all*)

lemma *dropD*: $\gamma,p\vdash \langle[Rule\ m\ Drop], Undecided\rangle \Rightarrow_\alpha t \Longrightarrow matches\ \gamma\ m\ Drop\ p \Longrightarrow t = Decision\ FinalDeny$
apply (*induction* [*Rule m Drop*] *Undecided t rule*: *approximating-bigstep-induct*)
by(*auto dest*: *skipD elim*!: *rules-singleton-rev-E*)

lemma *rejectD*: $\gamma,p\vdash \langle[Rule\ m\ Reject], Undecided\rangle \Rightarrow_\alpha t \Longrightarrow matches\ \gamma\ m\ Reject\ p \Longrightarrow t = Decision\ FinalDeny$
apply (*induction* [*Rule m Reject*] *Undecided t rule*: *approximating-bigstep-induct*)

by(*auto dest*: *skipD elim*!: *rules-singleton-rev-E*)

lemma *logD*: γ,*p*⊢ ⟨[*Rule m Log*], *Undecided*⟩ ⇒$_\alpha$ *t* ⟹ *t* = *Undecided*
apply (*induction* [*Rule m Log*] *Undecided t rule*: *approximating-bigstep-induct*)
by(*auto dest*: *skipD elim*!: *rules-singleton-rev-E*)

lemma *emptyD*: γ,*p*⊢ ⟨[*Rule m Empty*], *Undecided*⟩ ⇒$_\alpha$ *t* ⟹ *t* = *Undecided*
apply (*induction* [*Rule m Empty*] *Undecided t rule*: *approximating-bigstep-induct*)
by(*auto dest*: *skipD elim*!: *rules-singleton-rev-E*)

lemma *nomatchD*: γ,*p*⊢ ⟨[*Rule m a*], *Undecided*⟩ ⇒$_\alpha$ *t* ⟹ ¬ *matches* γ *m a p* ⟹ *t* = *Undecided*
apply (*induction* [*Rule m a*] *Undecided t rule*: *approximating-bigstep-induct*)
by(*auto dest*: *skipD elim*!: *rules-singleton-rev-E*)

lemmas *approximating-bigstepD* = *skipD acceptD dropD rejectD logD emptyD nomatchD decisionD*

lemma *approximating-bigstep-to-undecided*: γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *Undecided* ⟹ *s* = *Undecided*
by (*metis decisionD state.exhaust*)

lemma *approximating-bigstep-to-decision1*: γ,*p*⊢ ⟨*rs*, *Decision Y*⟩ ⇒$_\alpha$ *Decision X* ⟹ *Y* = *X*
by (*metis decisionD state.inject*)

lemma *nomatch-fst*: ¬ *matches* γ *m a p* ⟹ γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t* ⟹ γ,*p*⊢ ⟨*Rule m a* # *rs*, *s*⟩ ⇒$_\alpha$ *t*
apply(*cases s*)
apply(*clarify*)
apply(*drule nomatch*)
apply(*drule*(*1*) *seq*)
apply (*simp*; *fail*)
apply(*clarify*)
apply(*drule decisionD*)
apply(*clarify*)
apply(*simp add*: *decision*)
done

lemma *seq′*:
assumes *rs* = *rs*$_1$ @ *rs*$_2$ γ,*p*⊢ ⟨*rs*$_1$,*s*⟩ ⇒$_\alpha$ *t* γ,*p*⊢ ⟨*rs*$_2$,*t*⟩ ⇒$_\alpha$ *t′*
shows γ,*p*⊢ ⟨*rs*,*s*⟩ ⇒$_\alpha$ *t′*
using *assms* **by** (*cases s*) (*auto intro*: *seq decision dest*: *decisionD*)

lemma *seq-split*:
assumes γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t rs* = *rs*$_1$@*rs*$_2$
obtains *t′* **where** γ,*p*⊢ ⟨*rs*$_1$,*s*⟩ ⇒$_\alpha$ *t′* γ,*p*⊢ ⟨*rs*$_2$,*t′*⟩ ⇒$_\alpha$ *t*
using *assms*
proof (*induction rs s t arbitrary*: *rs*$_1$ *rs*$_2$ *thesis rule*: *approximating-bigstep-induct*)

case *Allow* **thus** *?case* **by** (*auto dest*: *skipD elim*!: *rules-singleton-rev-E intro*: *approximating-bigstep.intros*)
next
case *Deny* **thus** *?case* **by** (*auto dest*: *skipD elim*!: *rules-singleton-rev-E intro*: *approximating-bigstep.intros*)
next
case *Log* **thus** *?case* **by** (*auto dest*: *skipD elim*!: *rules-singleton-rev-E intro*: *approximating-bigstep.intros*)
next
case *Nomatch* **thus** *?case* **by** (*auto dest*: *skipD elim*!: *rules-singleton-rev-E intro*: *approximating-bigstep.intros*)
next
case (*Seq rs rsa rsb t t'*)
hence *rs*: *rsa* @ *rsb* = rs_1 @ rs_2 **by** *simp*
note *List.append-eq-append-conv-if*[*simp*]
from *rs* **show** *?case*
proof (*cases rule*: *list-app-eq-cases*)
case *longer*
with *Seq* **have** *t1*: γ,*p*⊢ ⟨*take* (*length rsa*) rs_1, *Undecided*⟩ ⇒$_\alpha$ *t*
by *simp*
from *Seq longer* **obtain** *t2*
where *t2a*: γ,*p*⊢ ⟨*drop* (*length rsa*) rs_1,*t*⟩ ⇒$_\alpha$ *t2*
and *rs2-t2*: γ,*p*⊢ ⟨rs_2,*t2*⟩ ⇒$_\alpha$ *t'*
by *blast*
with *t1 rs2-t2* **have** γ,*p*⊢ ⟨*take* (*length rsa*) rs_1 @ *drop* (*length rsa*) rs_1,*Undecided*⟩ ⇒$_\alpha$ *t2*
by (*blast intro*: *approximating-bigstep.seq*)
with *Seq rs2-t2* **show** *?thesis*
by *simp*
next
case *shorter*
with *rs* **have** *rsa'*: *rsa* = rs_1 @ *take* (*length rsa* − *length* rs_1) rs_2
by (*metis append-eq-conv-conj length-drop*)
from *shorter rs* **have** *rsb'*: *rsb* = *drop* (*length rsa* − *length* rs_1) rs_2
by (*metis append-eq-conv-conj length-drop*)
from *Seq rsa'* **obtain** *t1*
where *t1a*: γ,*p*⊢ ⟨rs_1,*Undecided*⟩ ⇒$_\alpha$ *t1*
and *t1b*: γ,*p*⊢ ⟨*take* (*length rsa* − *length* rs_1) rs_2,*t1*⟩ ⇒$_\alpha$ *t*
by *blast*
from *rsb' Seq.hyps* **have** *t2*: γ,*p*⊢ ⟨*drop* (*length rsa* − *length* rs_1) rs_2,*t*⟩ ⇒$_\alpha$ *t'*
by *blast*
with *seq' t1b* **have** γ,*p*⊢ ⟨rs_2,*t1*⟩ ⇒$_\alpha$ *t'* **by** (*metis append-take-drop-id*)
with *Seq t1a* **show** *?thesis*
by *fast*
qed
qed (*auto intro*: *approximating-bigstep.intros*)

lemma *seqE-fst*:
assumes γ,$p\vdash \langle r\#rs,\ s\rangle \Rightarrow_\alpha t$
obtains t' **where** γ,$p\vdash \langle[r],s\rangle \Rightarrow_\alpha t'$ γ,$p\vdash \langle rs,t'\rangle \Rightarrow_\alpha t$
using *assms seq-split* **by** (*metis append-Cons append-Nil*)

lemma *seq-fst*: **assumes** γ,$p\vdash \langle[r],\ s\rangle \Rightarrow_\alpha t$ **and** γ,$p\vdash \langle rs,\ t\rangle \Rightarrow_\alpha t'$ **shows** γ,$p\vdash \langle r\ \#\ rs,\ s\rangle \Rightarrow_\alpha t'$
proof(*cases s*)
case *Undecided* **with** *assms seq* **show** γ,$p\vdash \langle r\ \#\ rs,\ s\rangle \Rightarrow_\alpha t'$ **by** *fastforce*
next
case *Decision* **with** *assms* **show** γ,$p\vdash \langle r\ \#\ rs,\ s\rangle \Rightarrow_\alpha t'$
by(*auto simp*: *decision dest*!: *decisionD*)
qed

8.2 wf ruleset

A *'a rule list* here is well-formed (for a packet) if

- either the rules do not match
- or the action is not *Call*, not *Return*, not *Unknown*

definition *wf-ruleset* :: ('*a*, '*p*) *match-tac* ⇒ '*p* ⇒ '*a rule list* ⇒ *bool* **where**
wf-ruleset γ *p rs* ≡ ∀ *r* ∈ *set rs*.
(¬ *matches* γ (*get-match r*) (*get-action r*) *p*) ∨
(¬(∃ *chain*. *get-action r* = *Call chain*) ∧ *get-action r* ≠ *Return* ∧ ¬(∃ *chain*. *get-action r* = *Goto chain*) ∧ *get-action r* ≠ *Unknown*)

lemma *wf-ruleset-append*: *wf-ruleset* γ *p* (*rs1*@*rs2*) ⟷ *wf-ruleset* γ *p rs1* ∧ *wf-ruleset* γ *p rs2*
by(*auto simp add*: *wf-ruleset-def*)
lemma *wf-rulesetD*: **assumes** *wf-ruleset* γ *p* (*r* # *rs*) **shows** *wf-ruleset* γ *p* [*r*] **and** *wf-ruleset* γ *p rs*
using *assms* **by**(*auto simp add*: *wf-ruleset-def*)
lemma *wf-ruleset-fst*: *wf-ruleset* γ *p* (*Rule m a* # *rs*) ⟷ *wf-ruleset* γ *p* [*Rule m a*] ∧ *wf-ruleset* γ *p rs*
by(*auto simp add*: *wf-ruleset-def*)
lemma *wf-ruleset-stripfst*: *wf-ruleset* γ *p* (*r* # *rs*) ⟹ *wf-ruleset* γ *p* (*rs*)
by(*simp add*: *wf-ruleset-def*)
lemma *wf-ruleset-rest*: *wf-ruleset* γ *p* (*Rule m a* # *rs*) ⟹ *wf-ruleset* γ *p* [*Rule m a*]
by(*simp add*: *wf-ruleset-def*)

8.3 Ternary Semantics (Function)

fun *approximating-bigstep-fun* :: ('*a*, '*p*) *match-tac* ⇒ '*p* ⇒ '*a rule list* ⇒ *state* ⇒ *state* **where**
approximating-bigstep-fun γ *p* [] *s* = *s* |
approximating-bigstep-fun γ *p rs* (*Decision X*) = (*Decision X*) |

```
approximating-bigstep-fun γ p ((Rule m a)#rs) Undecided = (if
    ¬ matches γ m a p
  then
    approximating-bigstep-fun γ p rs Undecided
  else
    case a of Accept ⇒ Decision FinalAllow
            | Drop ⇒ Decision FinalDeny
            | Reject ⇒ Decision FinalDeny
            | Log ⇒ approximating-bigstep-fun γ p rs Undecided
            | Empty ⇒ approximating-bigstep-fun γ p rs Undecided
           — unhandled cases
           )
```

lemma *approximating-bigstep-fun-induct*[*case-names Empty Decision Nomatch Match*]
:
$(\bigwedge \gamma\ p\ s.\ P\ \gamma\ p\ []\ s) \Longrightarrow$
$(\bigwedge \gamma\ p\ r\ rs\ X.\ P\ \gamma\ p\ (r\ \#\ rs)\ (Decision\ X)) \Longrightarrow$
$(\bigwedge \gamma\ p\ m\ a\ rs.$
$\neg\ matches\ \gamma\ m\ a\ p \Longrightarrow P\ \gamma\ p\ rs\ Undecided \Longrightarrow P\ \gamma\ p\ (Rule\ m\ a\ \#\ rs)$
$Undecided) \Longrightarrow$
$(\bigwedge \gamma\ p\ m\ a\ rs.$
$matches\ \gamma\ m\ a\ p \Longrightarrow (a = Log \Longrightarrow P\ \gamma\ p\ rs\ Undecided) \Longrightarrow (a = Empty \Longrightarrow$
$P\ \gamma\ p\ rs\ Undecided) \Longrightarrow P\ \gamma\ p\ (Rule\ m\ a\ \#\ rs)\ Undecided) \Longrightarrow$
$P\ \gamma\ p\ rs\ s$
apply (*rule approximating-bigstep-fun.induct*[*of P γ p rs s*])
apply (*simp-all*)
by *metis*

lemma *Decision-approximating-bigstep-fun*: *approximating-bigstep-fun γ p rs* (*Decision*
X) = *Decision X*
by(*induction rs*) (*simp-all*)

lemma *approximating-bigstep-fun-induct-wf*[*case-names Empty Decision Nomatch*
MatchAccept MatchDrop MatchReject MatchLog MatchEmpty, *consumes 1*]:
wf-ruleset γ p rs $\Longrightarrow$
$(\bigwedge \gamma\ p\ s.\ P\ \gamma\ p\ []\ s) \Longrightarrow$
$(\bigwedge \gamma\ p\ r\ rs\ X.\ P\ \gamma\ p\ (r\ \#\ rs)\ (Decision\ X)) \Longrightarrow$
$(\bigwedge \gamma\ p\ m\ a\ rs.$
$\neg\ matches\ \gamma\ m\ a\ p \Longrightarrow P\ \gamma\ p\ rs\ Undecided \Longrightarrow P\ \gamma\ p\ (Rule\ m\ a\ \#\ rs)$
$Undecided) \Longrightarrow$
$(\bigwedge \gamma\ p\ m\ a\ rs.$
$matches\ \gamma\ m\ a\ p \Longrightarrow a = Accept\ \Longrightarrow P\ \gamma\ p\ (Rule\ m\ a\ \#\ rs)\ Undecided) \Longrightarrow$
$(\bigwedge \gamma\ p\ m\ a\ rs.$
$matches\ \gamma\ m\ a\ p \Longrightarrow a = Drop \Longrightarrow P\ \gamma\ p\ (Rule\ m\ a\ \#\ rs)\ Undecided) \Longrightarrow$
$(\bigwedge \gamma\ p\ m\ a\ rs.$

matches γ *m a p* $\Longrightarrow$ *a* = *Reject* $\Longrightarrow$ *P* γ *p* (*Rule m a # rs*) *Undecided*) $\Longrightarrow$
($\bigwedge\gamma$ *p m a rs.*
matches γ *m a p* $\Longrightarrow$ *a* = *Log* $\Longrightarrow$ *P* γ *p rs Undecided* $\Longrightarrow$ *P* γ *p* (*Rule m a # rs*) *Undecided*) $\Longrightarrow$
($\bigwedge\gamma$ *p m a rs.*
matches γ *m a p* $\Longrightarrow$ *a* = *Empty* $\Longrightarrow$ *P* γ *p rs Undecided* $\Longrightarrow$ *P* γ *p* (*Rule m a # rs*) *Undecided*) $\Longrightarrow$
P γ *p rs s*
proof(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)
case *Empty* **thus** *?case* **by** *blast*
next
case *Decision* **thus** *?case* **by** *blast*
next
case *Nomatch* **thus** *?case* **by**(*simp add*: *wf-ruleset-def*)
next
case (*Match* γ *p m a*) **thus** *?case*
apply −
apply(*frule wf-rulesetD*(*1*), *drule wf-rulesetD*(*2*))
apply(*simp*)
apply(*cases a*)
apply(*simp-all*)
apply(*auto simp add*: *wf-ruleset-def*)
done
qed

lemma *just-show-all-approximating-bigstep-fun-equalities-with-start-Undecided*[*case-names Undecided*]:
assumes *s* = *Undecided* $\Longrightarrow$ *approximating-bigstep-fun* γ *p rs1 s* = *approximating-bigstep-fun* γ *p rs2 s*
shows *approximating-bigstep-fun* γ *p rs1 s* = *approximating-bigstep-fun* γ *p rs2 s*
proof(*cases s*)
case *Undecided* **thus** *?thesis* **using** *assms* **by** *simp*
next
case *Decision* **thus** *?thesis* **by** (*simp add*: *Decision-approximating-bigstep-fun*)
qed

8.3.1 Append, Prepend, Postpend, Composition

lemma *approximating-bigstep-fun-seq-wf*: ⟦ *wf-ruleset* γ *p* rs_1⟧ $\Longrightarrow$
approximating-bigstep-fun γ *p* (rs_1 @ rs_2) *s* = *approximating-bigstep-fun* γ *p* rs_2 (*approximating-bigstep-fun* γ *p* rs_1 *s*)
proof(*induction* γ *p* rs_1 *s rule*: *approximating-bigstep-fun-induct*)
qed(*simp-all add*: *wf-ruleset-def Decision-approximating-bigstep-fun split*: *action.split*)

The state transitions from *Undecided* to *Undecided* if all intermediate states are *Undecided*

lemma *approximating-bigstep-fun-seq-Undecided-wf*: ⟦ *wf-ruleset* γ *p* (*rs1*@*rs2*)⟧

$\Longrightarrow$
approximating-bigstep-fun γ *p* (*rs1*@*rs2*) *Undecided* = *Undecided* $\longleftrightarrow$
approximating-bigstep-fun γ *p rs1 Undecided* = *Undecided* $\wedge$ *approximating-bigstep-fun* γ *p rs2 Undecided* = *Undecided*
proof(*induction* γ *p rs1 Undecided rule*: *approximating-bigstep-fun-induct*)
qed(*simp-all add*: *wf-ruleset-def split*: *action.split*)

lemma *approximating-bigstep-fun-seq-Undecided-t-wf*: $[\![$ *wf-ruleset* γ *p* (*rs1*@*rs2*)$]\!]$
$\Longrightarrow$
approximating-bigstep-fun γ *p* (*rs1*@*rs2*) *Undecided* = *t* $\longleftrightarrow$
approximating-bigstep-fun γ *p rs1 Undecided* = *Undecided* $\wedge$ *approximating-bigstep-fun* γ *p rs2 Undecided* = *t* $\vee$
approximating-bigstep-fun γ *p rs1 Undecided* = *t* $\wedge$ *t* $\neq$ *Undecided*
proof(*induction* γ *p rs1 Undecided rule*: *approximating-bigstep-fun-induct*)
case *Empty* **thus** *?case* **by**(*cases t*) *simp-all*
next
case *Nomatch* **thus** *?case* **by**(*simp add*: *wf-ruleset-def*)
next
case *Match* **thus** *?case* **by**(*auto simp add*: *wf-ruleset-def split*: *action.split*)
qed

lemma *approximating-bigstep-fun-wf-postpend*: *wf-ruleset* γ *p rsA* $\Longrightarrow$ *wf-ruleset* γ *p rsB* $\Longrightarrow$
approximating-bigstep-fun γ *p rsA s* = *approximating-bigstep-fun* γ *p rsB s*
$\Longrightarrow$
approximating-bigstep-fun γ *p* (*rsA*@*rsC*) *s* = *approximating-bigstep-fun* γ *p* (*rsB*@*rsC*) *s*
apply(*induction* γ *p rsA s rule*: *approximating-bigstep-fun-induct-wf*)
apply(*simp-all add*: *approximating-bigstep-fun-seq-wf*)
apply (*metis Decision-approximating-bigstep-fun*)+
done

lemma *approximating-bigstep-fun-singleton-prepend*:
assumes *approximating-bigstep-fun* γ *p rsB s* = *approximating-bigstep-fun* γ *p rsC s*
shows *approximating-bigstep-fun* γ *p* (*r*#*rsB*) *s* = *approximating-bigstep-fun* γ *p* (*r*#*rsC*) *s*
proof(*cases s*)
case *Decision* **thus** *?thesis* **by**(*simp add*: *Decision-approximating-bigstep-fun*)
next
case *Undecided*
with *assms* **show** *?thesis* **by**(*cases r*)(*simp split*: *action.split*)
qed

8.4 Equality with γ,$p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t$ semantics

lemma *approximating-bigstep-wf*: γ,$p\vdash \langle rs, Undecided\rangle \Rightarrow_\alpha Undecided \Longrightarrow$ *wf-ruleset* γ *p rs*
unfolding *wf-ruleset-def*
proof(*induction rs Undecided Undecided rule*: *approximating-bigstep-induct*)
case *Skip* **thus** *?case* **by** *simp*
next
case *Log* **thus** *?case* **by** *auto*
next
case *Nomatch* **thus** *?case* **by** *simp*
next
case (*Seq rs rs1 rs2 t*)
from *Seq approximating-bigstep-to-undecided* **have** $t = Undecided$ **by** *fast*
from *this Seq* **show** *?case* **by** *auto*
qed

only valid actions appear in this ruleset

definition *good-ruleset* :: *'a rule list* ⇒ *bool* **where**
good-ruleset rs ≡ ∀ *r* ∈ *set rs*. (¬(∃ *chain*. *get-action r* = *Call chain*) ∧ *get-action r* ≠ *Return* ∧ ¬(∃ *chain*. *get-action r* = *Goto chain*) ∧ *get-action r* ≠ *Unknown*)

lemma[*code*]: *good-ruleset rs* = (∀ *r*∈*set rs*. (*case get-action r of Call chain* ⇒ *False* | *Return* ⇒ *False* | *Goto chain* ⇒ *False* | *Unknown* ⇒ *False* | - ⇒ *True*))
unfolding *good-ruleset-def*
apply(*rule Set.ball-cong*)
apply(*simp-all*)
apply(*rename-tac r*)
by(*case-tac get-action r*)(*simp-all*)

lemma *good-ruleset-alt*: *good-ruleset rs* = (∀ *r*∈*set rs*. *get-action r* = *Accept* ∨ *get-action r* = *Drop* ∨ *get-action r* = *Reject* ∨ *get-action r* = *Log* ∨ *get-action r* = *Empty*)
unfolding *good-ruleset-def*
apply(*rule Set.ball-cong*)
apply(*simp-all*)
apply(*rename-tac r*)
by(*case-tac get-action r*)(*simp-all*)

lemma *good-ruleset-append*: *good-ruleset* (rs_1 @ rs_2) ⟷ *good-ruleset* rs_1 ∧ *good-ruleset* rs_2
by(*simp add*: *good-ruleset-alt*, *blast*)

lemma *good-ruleset-fst*: *good-ruleset* (*r*#*rs*) ⟹ *good-ruleset* [*r*]
by(*simp add*: *good-ruleset-def*)
lemma *good-ruleset-tail*: *good-ruleset* (*r*#*rs*) ⟹ *good-ruleset rs*
by(*simp add*: *good-ruleset-def*)

good-ruleset is stricter than *wf-ruleset*. It can be easily checked with running code!

lemma *good-imp-wf-ruleset*: *good-ruleset rs* $\Longrightarrow$ *wf-ruleset* γ *p rs* **by** (*metis good-ruleset-def wf-ruleset-def*)

lemma *simple-imp-good-ruleset*: *simple-ruleset rs* $\Longrightarrow$ *good-ruleset rs*
by(*simp add*: *simple-ruleset-def good-ruleset-def*, *fastforce*)

lemma *approximating-bigstep-fun-seq-semantics*: $[\![\gamma,p\vdash \langle rs_1,\ s\rangle \Rightarrow_\alpha t\ ]\!] \Longrightarrow$
approximating-bigstep-fun γ *p* $(rs_1$ @ $rs_2)$ *s* = *approximating-bigstep-fun* γ *p* rs_2 *t*
proof(*induction* rs_1 *s t arbitrary*: rs_2 *rule*: *approximating-bigstep.induct*)
qed(*simp-all add*: *Decision-approximating-bigstep-fun*)

lemma *approximating-semantics-imp-fun*: $\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t \Longrightarrow$ *approximating-bigstep-fun* γ *p rs s* = *t*
proof(*induction rs s t rule*: *approximating-bigstep-induct*)
qed(*auto simp add*: *approximating-bigstep-fun-seq-semantics Decision-approximating-bigstep-fun*)

lemma *approximating-fun-imp-semantics*: **assumes** *wf-ruleset* γ *p rs*
shows *approximating-bigstep-fun* γ *p rs s* = *t* $\Longrightarrow$ $\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t$
using *assms* **proof**(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct-wf*)
case (*Empty* γ *p s*)
thus $\gamma,p\vdash \langle[],\ s\rangle \Rightarrow_\alpha t$ **using** *skip* **by**(*simp*)
next
case (*Decision* γ *p r rs X*)
hence *t* = *Decision X* **by** *simp*
thus $\gamma,p\vdash \langle r$ # *rs*, *Decision X*$\rangle \Rightarrow_\alpha t$ **using** *decision* **by** *fast*
next
case (*Nomatch* γ *p m a rs*)
thus $\gamma,p\vdash \langle$*Rule m a* # *rs*, *Undecided*$\rangle \Rightarrow_\alpha t$
apply(*rule-tac t*=*Undecided* **in** *seq-fst*)
apply(*simp add*: *nomatch*)
apply(*simp add*: *Nomatch.IH*)
done
next
case (*MatchAccept* γ *p m a rs*)
hence *t* = *Decision FinalAllow* **by** *simp*
thus *?case* **by** (*metis MatchAccept.hyps accept decision seq-fst*)
next
case (*MatchDrop* γ *p m a rs*)
hence *t* = *Decision FinalDeny* **by** *simp*
thus *?case* **by** (*metis MatchDrop.hyps drop decision seq-fst*)
next
case (*MatchReject* γ *p m a rs*)
hence *t* = *Decision FinalDeny* **by** *simp*
thus *?case* **by** (*metis MatchReject.hyps reject decision seq-fst*)
next

case (*MatchLog* γ *p m a rs*)
thus *?case*
apply(*simp*)
apply(*rule-tac t=Undecided* **in** *seq-fst*)
apply(*simp add*: *log*)
apply(*simp add*: *MatchLog.IH*)
done
next
case (*MatchEmpty* γ *p m a rs*)
thus *?case*
apply(*simp*)
apply(*rule-tac t=Undecided* **in** *seq-fst*)
apply(*simp add*: *empty*)
apply(*simp add*: *MatchEmpty.IH*)
done
qed

Henceforth, we will use the *approximating-bigstep-fun* semantics, because they are easier. We show that they are equal.

theorem *approximating-semantics-iff-fun*: *wf-ruleset* γ *p rs* $\Longrightarrow$
γ,*p*$\vdash$ $\langle rs, s\rangle \Rightarrow_\alpha t \longleftrightarrow$ *approximating-bigstep-fun* γ *p rs s* = *t*
by (*metis approximating-fun-imp-semantics approximating-semantics-imp-fun*)

corollary *approximating-semantics-iff-fun-good-ruleset*: *good-ruleset rs* $\Longrightarrow$
γ,*p*$\vdash$ $\langle rs, s\rangle \Rightarrow_\alpha t \longleftrightarrow$ *approximating-bigstep-fun* γ *p rs s* = *t*
by (*metis approximating-semantics-iff-fun good-imp-wf-ruleset*)

lemma *approximating-bigstep-deterministic*: $[\![\gamma$,*p*$\vdash$ $\langle rs, s\rangle \Rightarrow_\alpha t$; γ,*p*$\vdash$ $\langle rs, s\rangle \Rightarrow_\alpha t']\!] \Longrightarrow t = t'$
proof(*induction arbitrary*: t' *rule*: *approximating-bigstep-induct*)
case *Seq* **thus** *?case*
by (*metis* (*opaque-lifting*, *mono-tags*) *append-Nil2 approximating-bigstep-fun.simps*(*1*) *approximating-bigstep-fun-seq-semantics*)
qed(*auto dest*: *approximating-bigstepD*)

lemma *rm-LogEmpty-fun-semantics*:
approximating-bigstep-fun γ *p* (*rm-LogEmpty rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)
case *Empty* **thus** *?case* **by**(*simp*)
next
case *Decision* **thus** *?case* **by**(*simp add*: *Decision-approximating-bigstep-fun*)
next
case (*Nomatch* γ *p m a rs*) **thus** *?case* **by**(*cases a*,*simp-all*)
next
case (*Match* γ *p m a rs*) **thus** *?case* **by**(*cases a*,*simp-all*)
qed

lemma γ,$p\vdash \langle$*rm-LogEmpty rs*, *s*$\rangle \Rightarrow_\alpha$ *t* $\longleftrightarrow$ γ,$p\vdash \langle$*rs*, *s*$\rangle \Rightarrow_\alpha$ *t*
apply(*rule iffI*)
apply(*induction rs arbitrary*: *s t*)
apply(*simp-all*)
apply(*rename-tac r rs s t*)
apply(*case-tac r*)
apply(*simp*)
apply(*rename-tac m a*)
apply(*case-tac a*)
apply(*simp-all*)
apply(*auto intro*: *approximating-bigstep.intros*)
apply(*erule seqE-fst*, *simp add*: *seq-fst*)
apply(*erule seqE-fst*, *simp add*: *seq-fst*)
apply (*metis decision log nomatch-fst seq-fst state.exhaust*)
apply(*erule seqE-fst*, *simp add*: *seq-fst*)
apply(*erule seqE-fst*, *simp add*: *seq-fst*)
apply(*erule seqE-fst*, *simp add*: *seq-fst*)
apply(*erule seqE-fst*, *simp add*: *seq-fst*)
apply (*metis decision empty nomatch-fst seq-fst state.exhaust*)
apply(*erule seqE-fst*, *simp add*: *seq-fst*)
apply(*induction rs s t rule*: *approximating-bigstep-induct*)
apply(*auto intro*: *approximating-bigstep.intros*)
apply(*rename-tac m a*)
apply(*case-tac a*)
apply(*auto intro*: *approximating-bigstep.intros*)
apply(*rename-tac* rs_1 rs_2 *t t'*)
apply(*drule-tac* rs_1=*rm-LogEmpty* rs_1 **and** rs_2=*rm-LogEmpty* rs_2 **in** *seq*)
apply(*simp-all*)
using *rm-LogEmpty-seq* **apply** *metis*
done

lemma *rm-LogEmpty-simple-but-Reject*:
good-ruleset rs $\Longrightarrow \forall$ *r* $\in$ *set* (*rm-LogEmpty rs*). *get-action r* = *Accept* $\vee$ *get-action r* = *Reject* $\vee$ *get-action r* = *Drop*
proof(*induction rs*)
case *Nil* **thus** *?case* **by**(*simp add*: *good-ruleset-def*)
next
case (*Cons r rs*) **thus** *?case*
apply(*clarify*)
apply(*cases r*, *rename-tac m a*, *simp*)
by(*case-tac a*) (*auto simp add*: *good-ruleset-def*)
qed

lemma *rw-Reject-fun-semantics*:
wf-unknown-match-tac $\alpha \Longrightarrow$
(*approximating-bigstep-fun* (β, α) *p* (*rw-Reject rs*) *s* = *approximating-bigstep-fun* (β, α) *p rs s*)
proof(*induction rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
thus *?case*
apply(*case-tac r*, *rename-tac m a*, *simp*)
apply(*case-tac a*)
apply(*case-tac* [!] *s*)
apply(*auto dest*: *wf-unknown-match-tacD-False1 wf-unknown-match-tacD-False2*)
done
qed

lemma *rmLogEmpty-rwReject-good-to-simple*: *good-ruleset rs* $\Longrightarrow$ *simple-ruleset* (*rw-Reject* (*rm-LogEmpty rs*))
apply(*drule rm-LogEmpty-simple-but-Reject*)
apply(*simp add*: *simple-ruleset-def*)
apply(*induction rs*)
apply(*simp-all*)
apply(*rename-tac r rs*)
apply(*case-tac r*)
apply(*rename-tac m a*)
apply(*case-tac a*)
apply(*simp-all*)
done

8.5 Matching

lemma *optimize-matches-option-generic*:
assumes $\forall$ *r* $\in$ *set rs*. *P* (*get-match r*) (*get-action r*)
and ($\bigwedge m\ m'\ a$. *P m a* $\Longrightarrow$ *f m* = *Some m'* $\Longrightarrow$ *matches* γ *m' a p* = *matches* γ *m a p*)
and ($\bigwedge m\ a$. *P m a* $\Longrightarrow$ *f m* = *None* $\Longrightarrow$ $\neg$ *matches* γ *m a p*)
shows *approximating-bigstep-fun* γ *p* (*optimize-matches-option f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
using *assms* **proof**(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)
case *Decision* **thus** *?case* **by** (*simp add*: *Decision-approximating-bigstep-fun*)
next
case (*Nomatch* γ *p m a rs*) **thus** *?case*
apply(*simp*)
apply(*cases f m*)
apply(*simp*; *fail*)
apply(*simp del*: *approximating-bigstep-fun.simps*)
apply(*rename-tac m'*)
apply(*subgoal-tac* $\neg$ *matches* γ *m' a p*)
apply(*simp*; *fail*)

using *assms* **by** *blast*
next
case (*Match* γ *p m a rs*) **thus** *?case*
apply(*cases f m*)
apply(*simp*; *fail*)
apply(*simp del*: *approximating-bigstep-fun.simps*)
apply(*rename-tac m'*)
apply(*subgoal-tac matches* γ *m' a p*)
apply(*simp split*: *action.split*; *fail*)
using *assms* **by** *blast*
qed(*simp*)

lemma *optimize-matches-generic*: $\forall$ *r* $\in$ *set rs. P* (*get-match r*) (*get-action r*) $\Longrightarrow$

($\bigwedge$*m a. P m a* $\Longrightarrow$ *matches* γ (*f m*) *a p* = *matches* γ *m a p*) $\Longrightarrow$
approximating-bigstep-fun γ *p* (*optimize-matches f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
unfolding *optimize-matches-def*
apply(*rule optimize-matches-option-generic*)
apply(*simp*; *fail*)
apply(*simp split*: *if-split-asm*)
apply *blast*
apply(*simp split*: *if-split-asm*)
using *matcheq-matchNone-not-matches* **by** *fast*

lemma *optimize-matches-matches-fst*: *matches* γ (*f m*) *a p* $\Longrightarrow$ *optimize-matches f* (*Rule m a* # *rs*) = (*Rule* (*f m*) *a*)# *optimize-matches f rs*
apply(*simp add*: *optimize-matches-def*)
by (*meson matcheq-matchNone-not-matches*)

lemma *optimize-matches*: $\forall$ *m a. matches* γ (*f m*) *a p* = *matches* γ *m a p* $\Longrightarrow$ *approximating-bigstep-fun* γ *p* (*optimize-matches f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
using *optimize-matches-generic*[**where** *P*=λ- -. *True*] **by** *metis*

lemma *optimize-matches-opt-MatchAny-match-expr*: *approximating-bigstep-fun* γ *p* (*optimize-matches opt-MatchAny-match-expr rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
using *optimize-matches opt-MatchAny-match-expr-correct* **by** *metis*

lemma *optimize-matches-a*: $\forall$ *a m. matches* γ *m a* = *matches* γ (*f a m*) *a* $\Longrightarrow$ *approximating-bigstep-fun* γ *p* (*optimize-matches-a f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)

case (*Match* γ *p m a rs*) **thus** *?case* **by**(*case-tac a*)(*simp-all add*: *optimize-matches-a-def*)
qed(*simp-all add*: *optimize-matches-a-def*)

lemma *optimize-matches-a-simplers*:
assumes *simple-ruleset rs* **and** $\forall$ *a m*. *a* = *Accept* $\vee$ *a* = *Drop* $\longrightarrow$ *matches* γ (*f a m*) *a* = *matches* γ *m a*
shows *approximating-bigstep-fun* γ *p* (*optimize-matches-a f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof −
from *assms*(*1*) **have** *wf-ruleset* γ *p rs* **by**(*simp add*: *simple-imp-good-ruleset good-imp-wf-ruleset*)
from ‹*wf-ruleset* γ *p rs*› *assms* **show** *approximating-bigstep-fun* γ *p* (*optimize-matches-a f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct-wf*)
case *Nomatch* **thus** *?case*
apply(*simp add*: *optimize-matches-a-def simple-ruleset-def*)
apply(*safe*)
apply(*simp-all*)
done
next
case *MatchReject* **thus** *?case* **by**(*simp add*: *optimize-matches-a-def simple-ruleset-def*)
qed(*simp-all add*: *optimize-matches-a-def simple-ruleset-tail*)
qed

lemma *not-matches-removeAll*: ¬ *matches* γ *m a p* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*removeAll* (*Rule m a*) *rs*) *Undecided* = *approximating-bigstep-fun* γ *p rs Undecided*
apply(*induction* γ *p rs Undecided rule*: *approximating-bigstep-fun.induct*)
apply(*simp*)
apply(*simp split*: *action.split*)
apply *blast*
done

end
theory *Datatype-Selectors*
imports *Main*
begin

Running Example: *datatype-new iptrule-match* = *is-Src*: *Src* (*src-range*: *ipt-iprange*)

A discriminator *disc* tells whether a value is of a certain constructor. Example: *is-Src*

A selector *sel* select the inner value. Example: *src-range*

A constructor *C* constructs a value Example: *Src*

The are well-formed if the belong together.

fun *wf-disc-sel* :: $((\'a \Rightarrow bool) \times (\'a \Rightarrow \'b)) \Rightarrow (\'b \Rightarrow \'a) \Rightarrow bool$ **where**
wf-disc-sel (*disc*, *sel*) *C* $\longleftrightarrow$ ($\forall$ *a*. *disc a* $\longrightarrow$ *C* (*sel a*) = *a*) $\wedge$ ($\forall$ *a*. ~~*disc a* $\longrightarrow$ *sel* (*C a*) = *a*~~)

declare *wf-disc-sel.simps*[*simp del*]

end
theory *IpAddresses*
imports *IP-Addresses.IP-Address-toString*
IP-Addresses.CIDR-Split
../Common/WordInterval-Lists
begin

— Misc

lemma *ipset-from-cidr* (*ipv4addr-of-dotdecimal* (*0*, *0*, *0*, *0*)) *33* = {*0*}
by(*simp add*: *ipv4addr-of-dotdecimal.simps ipv4addr-of-nat-def ipset-from-cidr-large-pfxlen*)

definition *all-but-those-ips* :: (*'i*::*len word* × *nat*) *list* ⇒ (*'i word* × *nat*) *list*
where
all-but-those-ips cidrips = *cidr-split* (*wordinterval-invert* (*l2wi* (*map ipcidr-to-interval cidrips*)))

lemma *all-but-those-ips*:
ipcidr-union-set (*set* (*all-but-those-ips cidrips*)) =
UNIV − (⋃ (*ip*,*n*) ∈ *set cidrips*. *ipset-from-cidr ip n*)
apply *simp*
unfolding *ipcidr-union-set-uncurry all-but-those-ips-def*
apply(*simp add*: *cidr-split-prefix*)
apply(*simp add*: *l2wi*)
apply(*simp add*: *ipcidr-to-interval-def*)
using *ipset-from-cidr-ipcidr-to-interval* **by** *blast*

9 IPv4 Addresses

9.1 IPv4 Addresses in IPTables Notation (how we parse it)

context
notes [[*typedef-overloaded*]]
begin
datatype *'i ipt-iprange* =
— Singleton IP Address

IpAddr 'i::len word

— CIDR notation: addr/xx
| *IpAddrNetmask 'i word nat*

— -m iprange –src-range a.b.c.d-e.f.g.h
| *IpAddrRange 'i word 'i word*

end

fun *ipt-iprange-to-set* :: *'i::len ipt-iprange* ⇒ *'i word set* **where**
ipt-iprange-to-set (*IpAddrNetmask base m*) = *ipset-from-cidr base m* |
ipt-iprange-to-set (*IpAddr ip*) = { *ip* } |
ipt-iprange-to-set (*IpAddrRange ip1 ip2*) = { *ip1* .. *ip2* }

ipt-iprange-to-set can only represent an empty set if it is an empty range.

lemma *ipt-iprange-to-set-nonempty*: *ipt-iprange-to-set ip* = {} ⟷
(∃ *ip1 ip2*. *ip* = *IpAddrRange ip1 ip2* ∧ *ip1* > *ip2*)
apply(*cases ip*)
apply(*simp*; *fail*)
apply(*simp add*: *ipset-from-cidr-alt bitmagic-zeroLast-leq-or1Last*; *fail*)
apply(*simp add*:*linorder-not-le*; *fail*)
done

maybe this is necessary as code equation?

context
includes *bit-operations-syntax*
begin

lemma *element-ipt-iprange-to-set*[*code-unfold*]: (*addr*::*'i*::*len word*) ∈ *ipt-iprange-to-set X* = (
case X of (*IpAddrNetmask pre len*) ⇒
(*pre AND* ((*mask len*) << (*len-of* (*TYPE*(*'i*)) − *len*))) ≤ *addr* ∧
addr ≤ *pre OR* (*mask* (*len-of* (*TYPE*(*'i*)) − *len*))
| *IpAddr ip* ⇒ (*addr* = *ip*)
| *IpAddrRange ip1 ip2* ⇒ *ip1* ≤ *addr* ∧ *ip2* ≥ *addr*)
apply(*cases X*)
apply(*simp*; *fail*)
apply(*simp add*: *ipset-from-cidr-alt*; *fail*)
apply(*simp*; *fail*)
done

end

lemma *ipt-iprange-to-set-uncurry-IpAddrNetmask*:
ipt-iprange-to-set (*uncurry IpAddrNetmask a*) = *uncurry ipset-from-cidr a*
by(*simp split*: *uncurry-splits*)

IP address ranges to (*start*, *end*) notation

fun *ipt-iprange-to-interval* :: *'i::len ipt-iprange* ⇒ (*'i word* × *'i word*) **where**
ipt-iprange-to-interval (*IpAddr addr*) = (*addr*, *addr*) |
ipt-iprange-to-interval (*IpAddrNetmask pre len*) = *ipcidr-to-interval* (*pre*, *len*) |
ipt-iprange-to-interval (*IpAddrRange ip1 ip2*) = (*ip1*, *ip2*)

lemma *ipt-iprange-to-interval*: *ipt-iprange-to-interval ip* = (*s*,*e*) ⟹ {*s* .. *e*} = *ipt-iprange-to-set ip*
apply(*cases ip*)
apply(*auto simp add*: *ipcidr-to-interval*)
done

A list of IP address ranges to a *'i wordinterval*. The nice thing is: the usual set operations are defined on this type. We can use the existing function *l2wi-intersect* if we want the intersection of the supplied list

lemma *wordinterval-to-set* (*l2wi-intersect* (*map ipt-iprange-to-interval ips*)) =
(⋂ *ip* ∈ *set ips*. *ipt-iprange-to-set ip*)
apply(*simp add*: *l2wi-intersect*)
using *ipt-iprange-to-interval* **by** *blast*

We can use *l2wi* if we want the union of the supplied list

lemma *wordinterval-to-set* (*l2wi* (*map ipt-iprange-to-interval ips*)) = (⋃ *ip* ∈ *set ips*. *ipt-iprange-to-set ip*)
apply(*simp add*: *l2wi*)
using *ipt-iprange-to-interval* **by** *blast*

A list of (negated) IP address to a *'i wordinterval*.

definition *ipt-iprange-negation-type-to-br-intersect* ::
'i::len ipt-iprange negation-type list ⇒ *'i wordinterval* **where**
ipt-iprange-negation-type-to-br-intersect l = *l2wi-negation-type-intersect* (*NegPos-map ipt-iprange-to-interval l*)

lemma *ipt-iprange-negation-type-to-br-intersect*: *wordinterval-to-set* (*ipt-iprange-negation-type-to-br-intersect l*) =
(⋂ *ip* ∈ *set* (*getPos l*). *ipt-iprange-to-set ip*) − (⋃ *ip* ∈ *set* (*getNeg l*). *ipt-iprange-to-set ip*)
apply(*simp add*: *ipt-iprange-negation-type-to-br-intersect-def l2wi-negation-type-intersect NegPos-map-simps*)
using *ipt-iprange-to-interval* **by** *blast*

The *'i wordinterval* can be translated back into a list of IP ranges. If a list of intervals is enough, we can use *wi2l*. If we need it in *'i ipt-iprange*, we can use this function.

definition *wi-2-cidr-ipt-iprange-list* :: *'i::len wordinterval* ⇒ *'i ipt-iprange list* **where**
wi-2-cidr-ipt-iprange-list r = *map* (*uncurry IpAddrNetmask*) (*cidr-split r*)

lemma *wi-2-cidr-ipt-iprange-list*:

```
(⋃ ip ∈ set (wi-2-cidr-ipt-iprange-list r). ipt-iprange-to-set ip) = wordinter-
val-to-set r
  proof −
  have (⋃ ip ∈ set (wi-2-cidr-ipt-iprange-list r). ipt-iprange-to-set ip) =
        (⋃x∈set (cidr-split r). uncurry ipset-from-cidr x)
    unfolding wi-2-cidr-ipt-iprange-list-def by force
  thus ?thesis using cidr-split-prefix by metis
 qed
```

For example, this allows the following transformation

```
 definition ipt-iprange-compress :: 'i::len ipt-iprange negation-type list ⇒ 'i ipt-iprange
list where
  ipt-iprange-compress = wi-2-cidr-ipt-iprange-list ∘ ipt-iprange-negation-type-to-br-intersect

 lemma ipt-iprange-compress: (⋃ ip ∈ set (ipt-iprange-compress l). ipt-iprange-to-set
ip) =
       (⋂ ip ∈ set (getPos l). ipt-iprange-to-set ip) − (⋃ ip ∈ set (getNeg l).
ipt-iprange-to-set ip)
  by (metis wi-2-cidr-ipt-iprange-list comp-apply ipt-iprange-compress-def ipt-iprange-negation-type-to-br-inter

 definition normalized-cidr-ip :: 'i::len ipt-iprange ⇒ bool where
  normalized-cidr-ip ip ≡ case ip of IpAddrNetmask - - ⇒ True | - ⇒ False

 lemma wi-2-cidr-ipt-iprange-list-normalized-IpAddrNetmask:
  ∀ a'∈set (wi-2-cidr-ipt-iprange-list as). normalized-cidr-ip a'
  apply(clarify)
  apply(simp add: wi-2-cidr-ipt-iprange-list-def normalized-cidr-ip-def)
  by force

 lemma ipt-iprange-compress-normalized-IpAddrNetmask:
  ∀ a'∈set (ipt-iprange-compress as). normalized-cidr-ip a'
 by(simp add: ipt-iprange-compress-def wi-2-cidr-ipt-iprange-list-normalized-IpAddrNetmask)



 definition ipt-iprange-to-cidr :: 'i::len ipt-iprange ⇒ ('i word × nat) list where
  ipt-iprange-to-cidr ips = cidr-split (iprange-interval (ipt-iprange-to-interval ips))

 lemma ipt-ipvange-to-cidr: ipcidr-union-set (set (ipt-iprange-to-cidr ips)) = (ipt-iprange-to-set
ips)
  apply(simp add: ipt-iprange-to-cidr-def)
  apply(simp add: ipcidr-union-set-uncurry)
  apply(case-tac (ipt-iprange-to-interval ips))
  apply(simp add: ipt-iprange-to-interval cidr-split-prefix-single)
  done
```

definition *interval-to-wi-to-ipt-iprange* :: *'i*::*len word* ⇒ *'i word* ⇒ *'i ipt-iprange*
where
interval-to-wi-to-ipt-iprange s e ≡
if s = *e*
then IpAddr s
else case cidr-split (*WordInterval s e*) *of* [(*ip*,*nmask*)] ⇒ *IpAddrNetmask ip nmask*
| - ⇒ *IpAddrRange s e*

lemma *interval-to-wi-to-ipt-ipv4range*: *ipt-iprange-to-set* (*interval-to-wi-to-ipt-iprange s e*) = {*s*..*e*}
proof −
from *cidr-split-prefix-single*[*of s e*] **have**
cidr-split (*WordInterval s e*) = [(*a*, *b*)] ⟹ *ipset-from-cidr a b* = {*s*..*e*} **for** *a b*
by(*simp add*: *iprange-interval.simps*)
thus *?thesis*
by(*simp add*: *interval-to-wi-to-ipt-iprange-def split*: *list.split*)
qed

fun *wi-to-ipt-iprange* :: *'i*::*len wordinterval* ⇒ *'i ipt-iprange list* **where**
wi-to-ipt-iprange (*WordInterval s e*) = (*if s* > *e then* [] *else*
[*interval-to-wi-to-ipt-iprange s e*]) |
wi-to-ipt-iprange (*RangeUnion a b*) = *wi-to-ipt-iprange a* @ *wi-to-ipt-iprange b*

lemma *wi-to-ipt-ipv4range*: ⋃(*set* (*map ipt-iprange-to-set* (*wi-to-ipt-iprange wi*))) = *wordinterval-to-set wi*
apply(*induction wi*)
apply(*simp add*: *interval-to-wi-to-ipt-ipv4range*)
apply(*simp*)
done

end
theory *L4-Protocol-Flags*
imports *Simple-Firewall.L4-Protocol*
begin

10 Matching TCP Flags

datatype *ipt-tcp-flags* = *TCP-Flags tcp-flag set* — mask
tcp-flag set — comp

definition *ipt-tcp-syn* :: *ipt-tcp-flags* **where**
ipt-tcp-syn ≡ *TCP-Flags* {*TCP-SYN*,*TCP-RST*,*TCP-ACK*,*TCP-FIN*} {*TCP-SYN*}

fun *match-tcp-flags* :: *ipt-tcp-flags* ⇒ *tcp-flag set* ⇒ *bool* **where**
match-tcp-flags (*TCP-Flags fmask c*) *flags* ⟷ (*flags* ∩ *fmask*) = *c*

lemma *match-tcp-flags ipt-tcp-syn* {*TCP-SYN*, *TCP-URG*, *TCP-PSH*} **by** *eval*

lemma *match-tcp-flags-nomatch*: ¬ *c* ⊆ *fmask* ⟹ ¬ *match-tcp-flags* (*TCP-Flags fmask c*) *pkt* **by** *auto*

definition *ipt-tcp-flags-NoMatch* :: *ipt-tcp-flags* **where**
ipt-tcp-flags-NoMatch ≡ *TCP-Flags* {} {*TCP-SYN*}
lemma *ipt-tcp-flags-NoMatch*: ¬ *match-tcp-flags ipt-tcp-flags-NoMatch pkt* **by**(*simp add*: *ipt-tcp-flags-NoMatch-def*)

definition *ipt-tcp-flags-Any* :: *ipt-tcp-flags* **where**
ipt-tcp-flags-Any ≡ *TCP-Flags* {} {}
lemma *ipt-tcp-flags-Any*: *match-tcp-flags ipt-tcp-flags-Any pkt* **by**(*simp add*: *ipt-tcp-flags-Any-def*)

lemma *ipt-tcp-flags-Any-isUNIV*: *fmask* = {} ∧ *c* = {} ⟷ (∀ *pkt*. *match-tcp-flags* (*TCP-Flags fmask c*) *pkt*) **by** *auto*

fun *match-tcp-flags-conjunct* :: *ipt-tcp-flags* ⇒ *ipt-tcp-flags* ⇒ *ipt-tcp-flags* **where**
match-tcp-flags-conjunct (*TCP-Flags fmask1 c1*) (*TCP-Flags fmask2 c2*) = (
if c1 ⊆ *fmask1* ∧ *c2* ⊆ *fmask2* ∧ *fmask1* ∩ *fmask2* ∩ *c1* = *fmask1* ∩ *fmask2* ∩ *c2*
then (*TCP-Flags* (*fmask1* ∪ *fmask2*) (*c1* ∪ *c2*))
else ipt-tcp-flags-NoMatch)

lemma *match-tcp-flags-conjunct*: *match-tcp-flags* (*match-tcp-flags-conjunct f1 f2*) *pkt* ⟷ *match-tcp-flags f1 pkt* ∧ *match-tcp-flags f2 pkt*
apply(*cases f1*, *cases f2*, *simp*)
apply(*rename-tac fmask1 c1 fmask2 c2*)
apply(*intro conjI impI*)
apply(*elim conjE*)
apply *blast*
apply(*simp add*: *ipt-tcp-flags-NoMatch*)
apply *fast*
done
declare *match-tcp-flags-conjunct.simps*[*simp del*]

Same as *match-tcp-flags-conjunct*, but returns *None* if result cannot match anyway

definition *match-tcp-flags-conjunct-option* :: *ipt-tcp-flags* ⇒ *ipt-tcp-flags* ⇒ *ipt-tcp-flags option* **where**
match-tcp-flags-conjunct-option f1 f2 = (*case match-tcp-flags-conjunct f1 f2 of* (*TCP-Flags fmask c*) ⇒ *if c* ⊆ *fmask then Some* (*TCP-Flags fmask c*) *else None*)

lemma *match-tcp-flags-conjunct-option ipt-tcp-syn* (*TCP-Flags* {*TCP-RST*, *TCP-ACK*} {*TCP-RST*}) = *None* **by** *eval*

lemma *match-tcp-flags-conjunct-option-Some*: *match-tcp-flags-conjunct-option f1 f2 = Some f3 ⟹*
match-tcp-flags f1 pkt ∧ match-tcp-flags f2 pkt ⟷ match-tcp-flags f3 pkt
apply(*simp add*: *match-tcp-flags-conjunct-option-def split*: *ipt-tcp-flags.split-asm if-split-asm*)
using *match-tcp-flags-conjunct* **by** *blast*
lemma *match-tcp-flags-conjunct-option-None*: *match-tcp-flags-conjunct-option f1 f2 = None ⟹*
¬(*match-tcp-flags f1 pkt ∧ match-tcp-flags f2 pkt*)
apply(*simp add*: *match-tcp-flags-conjunct-option-def split*: *ipt-tcp-flags.split-asm if-split-asm*)
using *match-tcp-flags-conjunct match-tcp-flags-nomatch* **by** *metis*

lemma *match-tcp-flags-conjunct-option*: (*case match-tcp-flags-conjunct-option f1 f2 of None ⇒ False | Some f3 ⇒ match-tcp-flags f3 pkt*) *⟷ match-tcp-flags f1 pkt ∧ match-tcp-flags f2 pkt*
apply(*simp split*: *option.split*)
using *match-tcp-flags-conjunct-option-Some match-tcp-flags-conjunct-option-None* **by** *blast*

fun *ipt-tcp-flags-equal* :: *ipt-tcp-flags ⇒ ipt-tcp-flags ⇒ bool* **where**
ipt-tcp-flags-equal (*TCP-Flags fmask1 c1*) (*TCP-Flags fmask2 c2*) = (
if c1 ⊆ fmask1 ∧ c2 ⊆ fmask2
then c1 = c2 ∧ fmask1 = fmask2
else (*¬ c1 ⊆ fmask1*) *∧* (*¬ c2 ⊆ fmask2*))
context
begin
private lemma *funny-set-falg-fmask-helper*: *c2 ⊆ fmask2 ⟹* (*c1 = c2 ∧ fmask1 = fmask2*) = (*∀ pkt.* (*pkt ∩ fmask1 = c1*) = (*pkt ∩ fmask2 = c2*))
apply *rule*
apply *presburger*
apply(*subgoal-tac fmask1 = fmask2*)
apply *blast*

proof −
assume *a1*: *c2 ⊆ fmask2*
assume *a2*: *∀ pkt.* (*pkt ∩ fmask1 = c1*) = (*pkt ∩ fmask2 = c2*)
have *f3*: *⋀A Aa.* (*A*::*'a set*) *− − Aa = Aa − − A*
by (*simp add*: *inf-commute*)
have *f4*: *⋀A Aa.* (*A*::*'a set*) *− −* (*− Aa*) = *A − Aa*
by *simp*
have *f5*: *⋀A Aa Ab.* (*A*::*'a set*) *− − Aa − − Ab = A − −* (*Aa − − Ab*)
by *blast*
have *f6*: *⋀A Aa.* (*A*::*'a set*) *−* (*− A − Aa*) = *A*
by *fastforce*
have *f7*: *⋀A Aa. − A*::*'a set*) *− − Aa = Aa − A*
using *f4 f3* **by** *presburger*

```
    have f8: ⋀A Aa. − (A::'a set) = − (A − Aa) − (A − − Aa)
      by blast
    have f9: c1 = − (− c1)
      by blast
    have f10: ⋀A. A − c1 − c1 = A − c1
      by blast
    have ⋀A. A − − (fmask1 − − fmask2) = c2 ∨ A − − fmask1 ≠ c1
      using f6 f5 a2 by (metis (no-types) Diff-Compl)
    hence f11: ⋀A. − A − − (fmask1 − − fmask2) = c2 ∨ fmask1 − A ≠ c1
      using f7 by meson
    have c2 − fmask2 = {}
      using a1 by force
    hence f12: − c2 − (fmask2 − c2) = − fmask2
      by blast
    hence fmask2 − − c2 = c2
      by blast
    hence f13: fmask1 − − c2 = c1
      using f3 a2 by simp
    hence f14: c1 = c2
      using f11 by blast
    hence f15: fmask2 − (fmask1 − c1) = c1
      using f13 f10 f9 f8 f7 f3 a2 by (metis Diff-Compl)
    have fmask1 − (fmask2 − c1) = c1
      using f14 f12 f10 f9 f8 f4 f3 a2 by (metis Diff-Compl)
    thus fmask1 = fmask2
      using f15 by blast
  qed

  lemma ipt-tcp-flags-equal: ipt-tcp-flags-equal f1 f2 ⟷ (∀ pkt. match-tcp-flags
f1 pkt = match-tcp-flags f2 pkt)
    apply(cases f1, cases f2, simp)
    apply(rename-tac fmask1 c1 fmask2 c2)
    apply(intro conjI impI)
     using funny-set-falg-fmask-helper apply metis
    apply blast
   done
 end
 declare ipt-tcp-flags-equal.simps[simp del]
end
theory Ports
imports
 HOL−Library.Word
 ../Common/WordInterval-Lists
 L4-Protocol-Flags
begin
```

11 Ports (layer 4)

E.g. source and destination ports for TCP/UDP

list of (start, end) port ranges

type-synonym *raw-ports* = (*16 word* × *16 word*) *list*

fun *ports-to-set* :: *raw-ports* ⇒ (*16 word*) *set* **where**
ports-to-set [] = {} |
ports-to-set ((*s*,*e*)#*ps*) = {*s*..*e*} ∪ *ports-to-set ps*

lemma *ports-to-set*: *ports-to-set pts* = ⋃ {{*s*..*e*} | *s e* . (*s*,*e*) ∈ *set pts*}
proof(*induction pts*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons p pts*) **thus** *?case* **by**(*cases p*, *simp*, *blast*)
qed

We can reuse the wordinterval theory to reason about ports

lemma *ports-to-set-wordinterval*: *ports-to-set ps* = *wordinterval-to-set* (*l2wi ps*)
by(*induction ps rule*: *l2wi.induct*) (*auto*)

inverting a raw listing of ports

definition *raw-ports-invert* :: *raw-ports* ⇒ *raw-ports* **where**
raw-ports-invert ps = *wi2l* (*wordinterval-invert* (*l2wi ps*))

lemma *raw-ports-invert*: *ports-to-set* (*raw-ports-invert ps*) = − *ports-to-set ps*
by(*auto simp add*: *raw-ports-invert-def l2wi-wi2l ports-to-set-wordinterval*)

A port always belongs to a protocol! We must not lose this information. You should never use *raw-ports* directly

datatype *ipt-l4-ports* = *L4Ports primitive-protocol raw-ports*

end
theory *Conntrack-State*
imports *../Common/Negation-Type Simple-Firewall.Lib-Enum-toString*
begin

datatype *ctstate* = *CT-New* | *CT-Established* | *CT-Related* | *CT-Untracked* | *CT-Invalid*

The state associated with a packet can be added as a tag to the packet. See `../Semantics_Stateful.thy`.

fun *match-ctstate* :: *ctstate set* ⇒ *ctstate* ⇒ *bool* **where**
match-ctstate S s-tag ⟷ *s-tag* ∈ *S*

fun *ctstate-conjunct* :: *ctstate set* ⇒ *ctstate set* ⇒ *ctstate set option* **where**

ctstate-conjunct S1 S2 = (*if S1* ∩ *S2* = {} *then None else Some* (*S1* ∩ *S2*))

value[*code*] *ctstate-conjunct* {*CT-Established*, *CT-New*} {*CT-New*}

lemma *ctstate-conjunct-correct*: *match-ctstate S1 pkt* ∧ *match-ctstate S2 pkt* ⟷

(*case ctstate-conjunct S1 S2 of None* ⇒ *False* | *Some S'* ⇒ *match-ctstate S' pkt*)
apply *simp*
by *blast*

lemma *UNIV-ctstate*: *UNIV* = {*CT-New*, *CT-Established*, *CT-Related*, *CT-Untracked*, *CT-Invalid*} **using** *ctstate.exhaust* **by** *auto*

instance *ctstate* :: *finite*
proof
from *UNIV-ctstate* **show** *finite* (*UNIV*:: *ctstate set*) **using** *finite.simps* **by** *auto*

qed

lemma *finite* (*S* :: *ctstate set*) **by** *simp*

instantiation *ctstate* :: *enum*
begin
definition *enum-ctstate* = [*CT-New*, *CT-Established*, *CT-Related*, *CT-Untracked*, *CT-Invalid*]

definition *enum-all-ctstate P* ⟷ *P CT-New* ∧ *P CT-Established* ∧ *P CT-Related* ∧ *P CT-Untracked* ∧ *P CT-Invalid*

definition *enum-ex-ctstate P* ⟷ *P CT-New* ∨ *P CT-Established* ∨ *P CT-Related* ∨ *P CT-Untracked* ∨ *P CT-Invalid*
instance proof
show *UNIV* = *set* (*enum-class.enum* :: *ctstate list*)
by(*simp add*: *UNIV-ctstate enum-ctstate-def*)
next
show *distinct* (*enum-class.enum* :: *ctstate list*)
by(*simp add*: *enum-ctstate-def*)
next
show ⋀*P*. (*enum-class.enum-all* :: (*ctstate* ⇒ *bool*) ⇒ *bool*) *P* = *Ball UNIV P*
by(*simp add*: *UNIV-ctstate enum-all-ctstate-def*)
next
show ⋀*P*. (*enum-class.enum-ex* :: (*ctstate* ⇒ *bool*) ⇒ *bool*) *P* = *Bex UNIV P*
by(*simp add*: *UNIV-ctstate enum-ex-ctstate-def*)
qed
end

definition *ctstate-is-UNIV* :: *ctstate set* ⇒ *bool* **where**
ctstate-is-UNIV c ≡ *CT-New* ∈ *c* ∧ *CT-Established* ∈ *c* ∧ *CT-Related* ∈ *c* ∧ *CT-Untracked* ∈ *c* ∧ *CT-Invalid* ∈ *c*

lemma *ctstate-is-UNIV*: *ctstate-is-UNIV c* ⟷ *c* = *UNIV*
unfolding *ctstate-is-UNIV-def*
apply(*simp add*: *UNIV-ctstate*)
apply(*rule iffI*)
apply(*clarify*)
using *UNIV-ctstate* **apply** *fastforce*
apply(*simp*)
done

value[*code*] *ctstate-is-UNIV* {*CT-Established*}

fun *ctstate-toString* :: *ctstate* ⇒ *string* **where**
ctstate-toString CT-New = *''NEW''* |
ctstate-toString CT-Established = *''ESTABLISHED''* |
ctstate-toString CT-Related = *''RELATED''* |
ctstate-toString CT-Untracked = *''UNTRACKED''* |
ctstate-toString CT-Invalid = *''INVALID''*

definition *ctstate-set-toString* :: *ctstate set* ⇒ *string* **where**
ctstate-set-toString S = *list-separated-toString ''*,*'' ctstate-toString* (*enum-set-to-list S*)

lemma *ctstate-set-toString* {*CT-New*, *CT-New*, *CT-Established*} = *''NEW*,*ESTABLISHED''*
by *eval*

end
theory *Tagged-Packet*
imports *Simple-Firewall.Simple-Packet Conntrack-State*
begin

12 Tagged Simple Packet

Packet constants are prefixed with *p*

A packet tagged with the following phantom fields: conntrack connection state

The idea to tag the connection state into the packet is sound. See `../Semantics_Stateful.thy`

record (**overloaded**) *'i tagged-packet* = *'i*::*len simple-packet* +
p-tag-ctstate :: *ctstate*

value (|
p-iiface = *''eth1''*, *p-oiface* = *''''*,
p-src = *0*, *p-dst* = *0*,
p-proto = *TCP*, *p-sport* = *0*, *p-dport* = *0*,
p-tcp-flags = {*TCP-SYN*},
p-payload = *''arbitrary payload''*,
p-tag-ctstate = *CT-New*
|):: *32 tagged-packet*

definition *simple-packet-tag*
:: *ctstate* ⇒ (*'i*::*len*, *'a*) *simple-packet-scheme* ⇒ (*'i*::*len*, *'a*) *tagged-packet-scheme*
where
simple-packet-tag ct-state p ≡
(|*p-iiface* = *p-iiface p*, *p-oiface* = *p-oiface p*, *p-src* = *p-src p*, *p-dst* = *p-dst p*,
p-proto = *p-proto p*,
p-sport = *p-sport p*, *p-dport* = *p-dport p*, *p-tcp-flags* = *p-tcp-flags p*,
p-payload = *p-payload p*,
p-tag-ctstate = *ct-state*,
... = *simple-packet.more p*|)

definition *tagged-packet-untag*
:: (*'i*::*len*, *'a*) *tagged-packet-scheme* ⇒ (*'i*::*len*, *'a*) *simple-packet-scheme* **where**
tagged-packet-untag p ≡
(|*p-iiface* = *p-iiface p*, *p-oiface* = *p-oiface p*, *p-src* = *p-src p*, *p-dst* = *p-dst p*,
p-proto = *p-proto p*,
p-sport = *p-sport p*, *p-dport* = *p-dport p*, *p-tcp-flags* = *p-tcp-flags p*,
p-payload = *p-payload p*,
... = *tagged-packet.more p*|)

lemma *tagged-packet-untag* (*simple-packet-tag ct-state p*) = *p*
simple-packet-tag ct-state (*tagged-packet-untag p*) = *p*(|*p-tag-ctstate* := *ct-state*|)
apply(*case-tac* [!] *p*)
by(*simp add*: *tagged-packet-untag-def simple-packet-tag-def*)+

end
theory *Common-Primitive-Syntax*
imports *../Datatype-Selectors*
IpAddresses
Simple-Firewall.Iface
L4-Protocol-Flags Ports Tagged-Packet Conntrack-State
begin

13 Primitive Matchers: Interfaces, IP Space, Layer 4 Ports Matcher

Primitive Match Conditions which only support interfaces, IPv4 addresses, layer 4 protocols, and layer 4 ports.

context
  notes [[*typedef-overloaded*]]
begin
  datatype *'i common-primitive* =
    is-Src: *Src* (*src-sel*: *'i*::*len ipt-iprange*) |
    is-Dst: *Dst* (*dst-sel*: *'i*::*len ipt-iprange*) |
    is-Iiface: *IIface* (*iiface-sel*: *iface*) |
    is-Oiface: *OIface* (*oiface-sel*: *iface*) |
    is-Prot: *Prot* (*prot-sel*: *protocol*) |
    is-Src-Ports: *Src-Ports* (*src-ports-sel*: *ipt-l4-ports*) |
    is-Dst-Ports: *Dst-Ports* (*dst-ports-sel*: *ipt-l4-ports*) |
    is-MultiportPorts: *MultiportPorts* (*multiportports-sel*: *ipt-l4-ports*) |
    is-L4-Flags: *L4-Flags* (*l4-flags-sel*: *ipt-tcp-flags*) |
    is-CT-State: *CT-State* (*ct-state-sel*: *ctstate set*) |
    is-Extra: *Extra* (*extra-sel*: *string*)
end

lemma *wf-disc-sel-common-primitive*:
      wf-disc-sel (*is-Src-Ports*, *src-ports-sel*) *Src-Ports*
      wf-disc-sel (*is-Dst-Ports*, *dst-ports-sel*) *Dst-Ports*
      wf-disc-sel (*is-Src*, *src-sel*) *Src*
      wf-disc-sel (*is-Dst*, *dst-sel*) *Dst*
      wf-disc-sel (*is-Iiface*, *iiface-sel*) *IIface*
      wf-disc-sel (*is-Oiface*, *oiface-sel*) *OIface*
      wf-disc-sel (*is-Prot*, *prot-sel*) *Prot*
      wf-disc-sel (*is-L4-Flags*, *l4-flags-sel*) *L4-Flags*
      wf-disc-sel (*is-CT-State*, *ct-state-sel*) *CT-State*
      wf-disc-sel (*is-Extra*, *extra-sel*) *Extra*
      wf-disc-sel (*is-MultiportPorts*, *multiportports-sel*) *MultiportPorts*
  by(*simp-all add*: *wf-disc-sel.simps*)

  — Example for a packet again:
  value (|*p-iiface* = *''eth0''*, *p-oiface* = *''eth1''*,
       p-src = *ipv4addr-of-dotdecimal* (*192*,*168*,*2*,*45*), *p-dst*= *ipv4addr-of-dotdecimal* (*173*,*194*,*112*,*111*),
          p-proto=*TCP*, *p-sport*=*2065*, *p-dport*=*80*, *p-tcp-flags* = {*TCP-ACK*},
          p-payload = *''GET / HTTP/1.0''*,
          p-tag-ctstate = *CT-Established*|) :: *32 tagged-packet*

end
theory *Unknown-Match-Tacs*
imports *Matching-Ternary*
begin

14 Approximate Matching Tactics

in-doubt-tactics

fun *in-doubt-allow* :: *'packet unknown-match-tac* **where**
  in-doubt-allow Accept - = True |
  in-doubt-allow Drop - = False |
  in-doubt-allow Reject - = False |
  in-doubt-allow - - = undefined

lemma *wf-in-doubt-allow*: *wf-unknown-match-tac in-doubt-allow*
  unfolding *wf-unknown-match-tac-def* **by**(*simp add*: *fun-eq-iff*)

fun *in-doubt-deny* :: *'packet unknown-match-tac* **where**
  in-doubt-deny Accept - = False |
  in-doubt-deny Drop - = True |
  in-doubt-deny Reject - = True |
  in-doubt-deny - - = undefined

lemma *wf-in-doubt-deny*: *wf-unknown-match-tac in-doubt-deny*
  unfolding *wf-unknown-match-tac-def* **by**(*simp add*: *fun-eq-iff*)

lemma *packet-independent-unknown-match-tacs*:
    packet-independent-α in-doubt-allow
    packet-independent-α in-doubt-deny
  by(*simp-all add*: *packet-independent-α-def*)

lemma *Drop-neq-Accept-unknown-match-tacs*:
      in-doubt-allow Drop ≠ in-doubt-allow Accept
      in-doubt-deny Drop ≠ in-doubt-deny Accept
  by(*simp-all add*: *fun-eq-iff*)

corollary *matches-induction-case-MatchNot-in-doubt-allow*:
∀ *a*. *matches* (*β*,*in-doubt-allow*) *m′ a p* = *matches* (*β*,*in-doubt-allow*) *m a p*
⟹
matches (*β*,*in-doubt-allow*) (*MatchNot m′*) *a p* = *matches* (*β*,*in-doubt-allow*) (*MatchNot m*) *a p*
by(*rule matches-induction-case-MatchNot*) (*simp-all add*: *Drop-neq-Accept-unknown-match-tacs packet-independent-unknown-match-tacs*)
corollary *matches-induction-case-MatchNot-in-doubt-deny*:
∀ *a*. *matches* (*β*,*in-doubt-deny*) *m′ a p* = *matches* (*β*,*in-doubt-deny*) *m a p*
⟹
matches (*β*,*in-doubt-deny*) (*MatchNot m′*) *a p* = *matches* (*β*,*in-doubt-deny*) (*MatchNot m*) *a p*
by(*rule matches-induction-case-MatchNot*) (*simp-all add*: *Drop-neq-Accept-unknown-match-tacs packet-independent-unknown-match-tacs*)

end
theory *Common-Primitive-Matcher-Generic*
imports *../Semantics-Ternary/Semantics-Ternary*
Common-Primitive-Syntax
../Semantics-Ternary/Unknown-Match-Tacs
begin

14.1 A Generic primitive matcher: Agnostic of IP Addresses

Generalized Definition agnostic of IP Addresses fro IPv4 and IPv6

locale *primitive-matcher-generic* =
fixes *β* :: (*'i*::*len common-primitive*, (*'i*::*len*, *'a*) *tagged-packet-scheme*) *exact-match-tac*
assumes *IIface*: ∀ *p i*. *β* (*IIface i*) *p* = *bool-to-ternary* (*match-iface i* (*p-iiface p*))
and *OIface*: ∀ *p i*. *β* (*OIface i*) *p* = *bool-to-ternary* (*match-iface i* (*p-oiface p*))
and *Prot*: ∀ *p proto*. *β* (*Prot proto*) *p* = *bool-to-ternary* (*match-proto proto* (*p-proto p*))
and *Src-Ports*: ∀ *p proto ps*. *β* (*Src-Ports* (*L4Ports proto ps*)) *p* = *bool-to-ternary* (*proto* = *p-proto p* ∧ *p-sport p* ∈ *ports-to-set ps*)
and *Dst-Ports*: ∀ *p proto ps*. *β* (*Dst-Ports* (*L4Ports proto ps*)) *p* = *bool-to-ternary* (*proto* = *p-proto p* ∧ *p-dport p* ∈ *ports-to-set ps*)
— -m multiport –ports matches source or destination port
and *MultiportsPorts*: ∀ *p proto ps*. *β* (*MultiportPorts* (*L4Ports proto ps*)) *p* = *bool-to-ternary* (*proto* = *p-proto p* ∧ (*p-sport p* ∈ *ports-to-set ps* ∨ *p-dport p* ∈ *ports-to-set ps*))
and *L4-Flags*: ∀ *p flags*. *β* (*L4-Flags flags*) *p* = *bool-to-ternary* (*match-tcp-flags flags* (*p-tcp-flags p*))
and *CT-State*: ∀ *p S*. *β* (*CT-State S*) *p* = *bool-to-ternary* (*match-ctstate S* (*p-tag-ctstate p*))
and *Extra*: ∀ *p str*. *β* (*Extra str*) *p* = *TernaryUnknown*
begin
lemma *Iface-single*:

matches (β, α) (*Match* (*IIface X*)) *a p* $\longleftrightarrow$ *match-iface X* (*p-iiface p*)
matches (β, α) (*Match* (*OIface X*)) *a p* $\longleftrightarrow$ *match-iface X* (*p-oiface p*)
by(*simp-all add*: *IIface OIface match-raw-ternary bool-to-ternary-simps split*: *ternaryvalue.split*)

Since matching on the iface cannot be *TernaryUnknown**, we can pull out negations.

lemma *Iface-single-not*:
matches (β, α) (*MatchNot* (*Match* (*IIface X*))) *a p* $\longleftrightarrow$ $\neg$ *match-iface X* (*p-iiface p*)
matches (β, α) (*MatchNot* (*Match* (*OIface X*))) *a p* $\longleftrightarrow$ $\neg$ *match-iface X* (*p-oiface p*)
by(*simp-all add*: *IIface OIface matches-case-ternaryvalue-tuple bool-to-ternary-simps split*: *ternaryvalue.split*)

lemma *Prot-single*:
matches (β, α) (*Match* (*Prot X*)) *a p* $\longleftrightarrow$ *match-proto X* (*p-proto p*)
by(*simp add*: *Prot match-raw-ternary bool-to-ternary-simps split*: *ternaryvalue.split*)
lemma *Prot-single-not*:
matches (β, α) (*MatchNot* (*Match* (*Prot X*))) *a p* $\longleftrightarrow$ $\neg$ *match-proto X* (*p-proto p*)
by(*simp add*: *Prot matches-case-ternaryvalue-tuple bool-to-ternary-simps split*: *ternaryvalue.split*)

lemma *Ports-single*:
matches (β, α) (*Match* (*Src-Ports* (*L4Ports proto ps*))) *a p* $\longleftrightarrow$ *proto* = *p-proto p* $\wedge$ *p-sport p* $\in$ *ports-to-set ps*
matches (β, α) (*Match* (*Dst-Ports* (*L4Ports proto ps*))) *a p* $\longleftrightarrow$ *proto* = *p-proto p* $\wedge$ *p-dport p* $\in$ *ports-to-set ps*
by(*simp-all add*: *Src-Ports Dst-Ports match-raw-ternary bool-to-ternary-simps split*: *ternaryvalue.split*)
lemma *Ports-single-not*:
matches (β, α) (*MatchNot* (*Match* (*Src-Ports* (*L4Ports proto ps*)))) *a p* $\longleftrightarrow$ *proto* $\neq$ *p-proto p* $\vee$ *p-sport p* $\notin$ *ports-to-set ps*
matches (β, α) (*MatchNot* (*Match* (*Dst-Ports* (*L4Ports proto ps*)))) *a p* $\longleftrightarrow$ *proto* $\neq$ *p-proto p* $\vee$ *p-dport p* $\notin$ *ports-to-set ps*
by(*simp-all add*: *Src-Ports Dst-Ports matches-case-ternaryvalue-tuple bool-to-ternary-simps split*: *ternaryvalue.split*)

Ports are dependent matches. They always match on the protocol too

lemma *Ports-single-rewrite-Prot*:
matches (β, α) (*Match* (*Src-Ports* (*L4Ports proto ps*))) *a p* $\longleftrightarrow$
matches (β, α) (*Match* (*Prot* (*Proto proto*))) *a p* $\wedge$ *p-sport p* $\in$ *ports-to-set ps*
matches (β, α) (*MatchNot* (*Match* (*Src-Ports* (*L4Ports proto ps*)))) *a p* $\longleftrightarrow$
matches (β, α) (*MatchNot* (*Match* (*Prot* (*Proto proto*)))) *a p* $\vee$ *p-sport p* $\notin$ *ports-to-set ps*
matches (β, α) (*Match* (*Dst-Ports* (*L4Ports proto ps*))) *a p* $\longleftrightarrow$
matches (β, α) (*Match* (*Prot* (*Proto proto*))) *a p* $\wedge$ *p-dport p* $\in$ *ports-to-set ps*

matches (*β*, *α*) (*MatchNot* (*Match* (*Dst-Ports* (*L4Ports proto ps*)))) *a p* ⟷
matches (*β*, *α*) (*MatchNot* (*Match* (*Prot* (*Proto proto*)))) *a p* ∨ *p-dport p* ∉ *ports-to-set ps*
by(*auto simp add*: *Ports-single-not Ports-single Prot-single-not Prot-single*)

lemma *multiports-disjuction*:
(∃ *rg*∈*set spts*. *matches* (*β*, *α*) (*Match* (*Src-Ports* (*L4Ports proto* [*rg*]))) *a p*) ⟷ *matches* (*β*, *α*) (*Match* (*Src-Ports* (*L4Ports proto spts*))) *a p*
(∃ *rg*∈*set dpts*. *matches* (*β*, *α*) (*Match* (*Dst-Ports* (*L4Ports proto* [*rg*]))) *a p*) ⟷ *matches* (*β*, *α*) (*Match* (*Dst-Ports* (*L4Ports proto dpts*))) *a p*
by(*auto simp add*: *Src-Ports Dst-Ports match-raw-ternary bool-to-ternary-simps ports-to-set*
split: *ternaryvalue.split*)

lemma *MultiportPorts-single-rewrite*:
matches (*β*, *α*) (*Match* (*MultiportPorts ports*)) *a p* ⟷
matches (*β*, *α*) (*Match* (*Src-Ports ports*)) *a p* ∨ *matches* (*β*, *α*) (*Match* (*Dst-Ports ports*)) *a p*
apply(*cases ports*)
apply(*simp add*: *Ports-single*)
by(*simp add*: *MultiportsPorts match-raw-ternary bool-to-ternary-simps*
split: *ternaryvalue.split*)
lemma *MultiportPorts-single-rewrite-MatchOr*:
matches (*β*, *α*) (*Match* (*MultiportPorts ports*)) *a p* ⟷
matches (*β*, *α*) (*MatchOr* (*Match* (*Src-Ports ports*)) (*Match* (*Dst-Ports ports*))) *a p*
apply(*cases ports*)
by(*simp add*: *MatchOr MultiportPorts-single-rewrite*)

lemma *MultiportPorts-single-not-rewrite-MatchAnd*:
matches (*β*, *α*) (*MatchNot* (*Match* (*MultiportPorts ports*))) *a p* ⟷
matches (*β*, *α*) (*MatchAnd* (*MatchNot* (*Match* (*Src-Ports ports*))) (*MatchNot* (*Match* (*Dst-Ports ports*)))) *a p*
apply(*cases ports*)
apply(*simp add*: *Ports-single-not bunch-of-lemmata-about-matches*)
by(*simp add*: *MultiportsPorts matches-case-ternaryvalue-tuple bool-to-ternary-simps*
split: *ternaryvalue.split*)
lemma *MultiportPorts-single-not-rewrite*:
matches (*β*, *α*) (*MatchNot* (*Match* (*MultiportPorts ports*))) *a p* ⟷
¬ *matches* (*β*, *α*) (*Match* (*Src-Ports ports*)) *a p* ∧ ¬ *matches* (*β*, *α*) (*Match* (*Dst-Ports ports*)) *a p*
apply(*cases ports*)
by(*simp add*: *MultiportPorts-single-not-rewrite-MatchAnd bunch-of-lemmata-about-matches*
Ports-single-not Ports-single)

lemma *Extra-single*:
matches (*β*, *α*) (*Match* (*Extra str*)) *a p* ⟷ *α a p*

by(*simp add*: *Extra match-raw-ternary*)
lemma *Extra-single-not*: — ternary logic, ¬ *unknown* = *unknown*
matches (*β*, *α*) (*MatchNot* (*Match* (*Extra str*))) *a p* ⟷ *α a p*
by(*simp add*: *Extra matches-case-ternaryvalue-tuple*)
end

14.2 Basic optimisations

Compress many *Extra* expressions to one expression.

fun *compress-extra* :: *'i*::*len common-primitive match-expr* ⇒ *'i common-primitive match-expr* **where**
compress-extra (*Match x*) = *Match x* |
compress-extra (*MatchNot* (*Match* (*Extra e*))) = *Match* (*Extra* (*''NOT* (*''@e@''*)*''*)) |
compress-extra (*MatchNot m*) = (*MatchNot* (*compress-extra m*)) |

compress-extra (*MatchAnd* (*Match* (*Extra e1*)) *m2*) = (*case compress-extra m2 of Match* (*Extra e2*) ⇒ *Match* (*Extra* (*e1@'' ''@e2*)) | *MatchAny* ⇒ *Match* (*Extra e1*) | *m2'* ⇒ *MatchAnd* (*Match* (*Extra e1*)) *m2'*) |
compress-extra (*MatchAnd m1 m2*) = *MatchAnd* (*compress-extra m1*) (*compress-extra m2*) |

compress-extra MatchAny = *MatchAny*

thm *compress-extra.simps*

value [*nbe*] *compress-extra* (*MatchAnd* (*Match* (*Extra ''foo''*)) (*Match* (*Extra ''bar''*)))
value [*nbe*] *compress-extra* (*MatchAnd* (*Match* (*Extra ''foo''*)) (*MatchNot* (*Match* (*Extra ''bar''*))))
value [*nbe*] *compress-extra* (*MatchAnd* (*Match* (*Extra ''−m''*)) (*MatchAnd* (*Match* (*Extra ''addrtype''*)) (*MatchAnd* (*Match* (*Extra ''−−dst−type''*)) (*MatchAnd* (*Match* (*Extra ''BROADCAST''*)) *MatchAny*))))

lemma *compress-extra-correct-matchexpr*:
fixes *β*::(*'i*::*len common-primitive*, (*'i*::*len*, *'a*) *tagged-packet-scheme*) *exact-match-tac*
assumes *generic*: *primitive-matcher-generic β*
shows *matches* (*β*, *α*) *m* = *matches* (*β*, *α*) (*compress-extra m*)
proof(*simp add*: *fun-eq-iff*, *clarify*, *rename-tac a p*)
fix *a* **and** *p* :: (*'i*, *'a*) *tagged-packet-scheme*
from *generic* **have** *β* (*Extra e*) *p* = *TernaryUnknown* **for** *e* **by**(*simp add*: *primitive-matcher-generic.Extra*)
hence *ternary-ternary-eval* (*map-match-tac β p m*) = *ternary-ternary-eval* (*map-match-tac β p* (*compress-extra m*))
proof(*induction m rule*: *compress-extra.induct*)
case *4* **thus** *?case*
by(*simp-all split*: *match-expr.split match-expr.split-asm common-primitive.split*)
qed (*simp-all*)

thus *matches* (β, α) *m a p* = *matches* (β, α) (*compress-extra m*) *a p*
by(*rule matches-iff-apply-f*)
qed

end
theory *Common-Primitive-Matcher*
imports *Common-Primitive-Matcher-Generic*
begin

14.3 Primitive Matchers: IP Port Iface Matcher

fun *common-matcher* :: ('*i*::*len common-primitive*, ('*i*, '*a*) *tagged-packet-scheme*)
exact-match-tac **where**
common-matcher (*IIface i*) *p* = *bool-to-ternary* (*match-iface i* (*p-iiface p*)) |
common-matcher (*OIface i*) *p* = *bool-to-ternary* (*match-iface i* (*p-oiface p*)) |

common-matcher (*Src ip*) *p* = *bool-to-ternary* (*p-src p* ∈ *ipt-iprange-to-set ip*) |
common-matcher (*Dst ip*) *p* = *bool-to-ternary* (*p-dst p* ∈ *ipt-iprange-to-set ip*) |

common-matcher (*Prot proto*) *p* = *bool-to-ternary* (*match-proto proto* (*p-proto p*)) |

common-matcher (*Src-Ports* (*L4Ports proto ps*)) *p* = *bool-to-ternary* (*proto* = *p-proto p* ∧ *p-sport p* ∈ *ports-to-set ps*) |
common-matcher (*Dst-Ports* (*L4Ports proto ps*)) *p* = *bool-to-ternary* (*proto* = *p-proto p* ∧ *p-dport p* ∈ *ports-to-set ps*) |

common-matcher (*MultiportPorts* (*L4Ports proto ps*)) *p* = *bool-to-ternary* (*proto* = *p-proto p* ∧ (*p-sport p* ∈ *ports-to-set ps* ∨ *p-dport p* ∈ *ports-to-set ps*)) |

common-matcher (*L4-Flags flags*) *p* = *bool-to-ternary* (*match-tcp-flags flags* (*p-tcp-flags p*)) |

common-matcher (*CT-State S*) *p* = *bool-to-ternary* (*match-ctstate S* (*p-tag-ctstate p*)) |

common-matcher (*Extra* -) *p* = *TernaryUnknown*

lemma *packet-independent-*β*-unknown-common-matcher*: *packet-independent-*β*-unknown common-matcher*
apply(*simp add*: *packet-independent-*β*-unknown-def*)
apply(*clarify*)
apply(*rename-tac a p1 p2*)
apply(*case-tac a*)
apply(*simp-all add*: *bool-to-ternary-Unknown*)
apply(*rename-tac l4ports*, *case-tac l4ports*; *simp add*: *bool-to-ternary-Unknown*; *fail*)+

done

lemma *primitive-matcher-generic-common-matcher*: *primitive-matcher-generic common-matcher*
by *unfold-locales simp-all*

Warning: beware of the sloppy term 'empty' portrange

An 'empty' port range means it can never match! Basically, *MatchNot* (*Match* (*Src-Ports* (*L4Ports proto* [(*0*, *65535*)]))) is False

lemma ¬ *matches* (*common-matcher*, *α*) (*MatchNot* (*Match* (*Src-Ports* (*L4Ports TCP* [(*0*,*65535*)])))) *a*
⟦*p-iiface* = *''eth0''*, *p-oiface* = *''eth1''*,
p-src = *ipv4addr-of-dotdecimal* (*192*,*168*,*2*,*45*), *p-dst*= *ipv4addr-of-dotdecimal* (*173*,*194*,*112*,*111*),
p-proto=*TCP*, *p-sport*=*2065*, *p-dport*=*80*, *p-tcp-flags* = {},
p-payload = *''''*, *p-tag-ctstate* = *CT-New*⟧
by(*simp add*: *primitive-matcher-generic-common-matcher primitive-matcher-generic.Ports-single-not*)

An 'empty' port range means it always matches! Basically, *MatchNot* (*Match* (*Src-Ports* (*L4Ports any* []))) is True. This corresponds to firewall behavior, but usually you cannot specify an empty portrange in firewalls, but omission of portrange means no-port-restrictions, i.e. every port matches.

lemma *matches* (*common-matcher*, *α*) (*MatchNot* (*Match* (*Src-Ports* (*L4Ports any* [])))) *a*
⟦*p-iiface* = *''eth0''*, *p-oiface* = *''eth1''*,
p-src = *ipv4addr-of-dotdecimal* (*192*,*168*,*2*,*45*), *p-dst*= *ipv4addr-of-dotdecimal* (*173*,*194*,*112*,*111*),
p-proto=*TCP*, *p-sport*=*2065*, *p-dport*=*80*, *p-tcp-flags* = {},
p-payload = *''''*, *p-tag-ctstate* = *CT-New*⟧
by(*simp add*: *primitive-matcher-generic-common-matcher primitive-matcher-generic.Ports-single-not*)

If not a corner case, portrange matching is straight forward.

lemma *matches* (*common-matcher*, *α*) (*Match* (*Src-Ports* (*L4Ports TCP* [(*1024*,*4096*), (*9999*, *65535*)]))) *a*
⟦*p-iiface* = *''eth0''*, *p-oiface* = *''eth1''*,
p-src = *ipv4addr-of-dotdecimal* (*192*,*168*,*2*,*45*), *p-dst*= *ipv4addr-of-dotdecimal* (*173*,*194*,*112*,*111*),
p-proto=*TCP*, *p-sport*=*2065*, *p-dport*=*80*, *p-tcp-flags* = {},
p-payload = *''''*, *p-tag-ctstate* = *CT-New*⟧
¬ *matches* (*common-matcher*, *α*) (*Match* (*Src-Ports* (*L4Ports TCP* [(*1024*,*4096*), (*9999*, *65535*)]))) *a*
⟦*p-iiface* = *''eth0''*, *p-oiface* = *''eth1''*,
p-src = *ipv4addr-of-dotdecimal* (*192*,*168*,*2*,*45*), *p-dst*= *ipv4addr-of-dotdecimal* (*173*,*194*,*112*,*111*),
p-proto=*TCP*, *p-sport*=*5000*, *p-dport*=*80*, *p-tcp-flags* = {},
p-payload = *''''*, *p-tag-ctstate* = *CT-New*⟧
¬*matches* (*common-matcher*, *α*) (*MatchNot* (*Match* (*Src-Ports* (*L4Ports TCP* [(*1024*,*4096*), (*9999*, *65535*)])))) *a*

(⦇p-iiface = ''eth0'', p-oiface = ''eth1'',
p-src = ipv4addr-of-dotdecimal (192,168,2,45), p-dst= ipv4addr-of-dotdecimal (173,194,112,111),
p-proto=TCP, p-sport=2065, p-dport=80, p-tcp-flags = {},
p-payload = '''', p-tag-ctstate = CT-New⦈)
by(*simp-all add*: *primitive-matcher-generic-common-matcher primitive-matcher-generic.Ports-single-not primitive-matcher-generic.Ports-single*)

Lemmas when matching on *Src* or *Dst*

lemma *common-matcher-SrcDst-defined*:
common-matcher (*Src m*) *p* ≠ *TernaryUnknown*
common-matcher (*Dst m*) *p* ≠ *TernaryUnknown*
common-matcher (*Src-Ports ps*) *p* ≠ *TernaryUnknown*
common-matcher (*Dst-Ports ps*) *p* ≠ *TernaryUnknown*
common-matcher (*MultiportPorts ps*) *p* ≠ *TernaryUnknown*
apply(*case-tac* [!] *m*, *case-tac* [!] *ps*)
apply(*simp-all add*: *bool-to-ternary-Unknown*)
done
lemma *common-matcher-SrcDst-defined-simp*:
common-matcher (*Src x*) *p* ≠ *TernaryFalse* ⟷ *common-matcher* (*Src x*) *p* = *TernaryTrue*
common-matcher (*Dst x*) *p* ≠ *TernaryFalse* ⟷ *common-matcher* (*Dst x*) *p* = *TernaryTrue*
apply (*metis eval-ternary-Not.cases common-matcher-SrcDst-defined*(*1*) *ternary-value.distinct*(*1*))
apply (*metis eval-ternary-Not.cases common-matcher-SrcDst-defined*(*2*) *ternary-value.distinct*(*1*))
done

lemma *match-simplematcher-SrcDst*:
matches (*common-matcher*, α) (*Match* (*Src X*)) *a p* ⟷ *p-src p* ∈ *ipt-iprange-to-set X*
matches (*common-matcher*, α) (*Match* (*Dst X*)) *a p* ⟷ *p-dst p* ∈ *ipt-iprange-to-set X*
by(*simp-all add*: *match-raw-ternary bool-to-ternary-simps split*: *ternaryvalue.split*)
lemma *match-simplematcher-SrcDst-not*:
matches (*common-matcher*, α) (*MatchNot* (*Match* (*Src X*))) *a p* ⟷ *p-src p* ∉ *ipt-iprange-to-set X*
matches (*common-matcher*, α) (*MatchNot* (*Match* (*Dst X*))) *a p* ⟷ *p-dst p* ∉ *ipt-iprange-to-set X*
apply(*simp-all add*: *matches-case-ternaryvalue-tuple split*: *ternaryvalue.split*)
apply(*case-tac* [!] *X*)
apply(*simp-all add*: *bool-to-ternary-simps*)
done
lemma *common-matcher-SrcDst-Inter*:
(∀ *m*∈*set X*. *matches* (*common-matcher*, α) (*Match* (*Src m*)) *a p*) ⟷ *p-src p* ∈ (⋂ *x*∈*set X*. *ipt-iprange-to-set x*)
(∀ *m*∈*set X*. *matches* (*common-matcher*, α) (*Match* (*Dst m*)) *a p*) ⟷ *p-dst p*

$\in (\bigcap x \in set\ X.\ ipt\text{-}iprange\text{-}to\text{-}set\ x)$
by(*simp-all add*: *match-raw-ternary bool-to-ternary-simps split*: *ternaryvalue.split*)

14.4 Basic optimisations

Perform very basic optimization. Remove matches to primitives which are essentially *MatchAny*

fun *optimize-primitive-univ* :: *'i::len common-primitive match-expr* ⇒ *'i common-primitive match-expr* **where**
optimize-primitive-univ (*Match* (*Src* (*IpAddrNetmask - 0*))) = *MatchAny* |
optimize-primitive-univ (*Match* (*Dst* (*IpAddrNetmask - 0*))) = *MatchAny* |

optimize-primitive-univ (*Match* (*IIface iface*)) = (*if iface = ifaceAny then MatchAny else* (*Match* (*IIface iface*))) |
optimize-primitive-univ (*Match* (*OIface iface*)) = (*if iface = ifaceAny then MatchAny else* (*Match* (*OIface iface*))) |

optimize-primitive-univ (*Match* (*Prot ProtoAny*)) = *MatchAny* |
optimize-primitive-univ (*Match* (*L4-Flags* (*TCP-Flags m c*))) = (*if m* = {} ∧ *c* = {} *then MatchAny else* (*Match* (*L4-Flags* (*TCP-Flags m c*)))) |
optimize-primitive-univ (*Match* (*CT-State ctstate*)) = (*if ctstate-is-UNIV ctstate then MatchAny else Match* (*CT-State ctstate*)) |
optimize-primitive-univ (*Match m*) = *Match m* |

optimize-primitive-univ (*MatchNot m*) = (*MatchNot* (*optimize-primitive-univ m*)) |

optimize-primitive-univ (*MatchAnd m1 m2*) = *MatchAnd* (*optimize-primitive-univ m1*) (*optimize-primitive-univ m2*) |
optimize-primitive-univ MatchAny = *MatchAny*

lemma *optimize-primitive-univ-unchanged-primitives*:
optimize-primitive-univ (*Match a*) = (*Match a*) ∨ *optimize-primitive-univ* (*Match a*) = *MatchAny*
by (*induction* (*Match a*) *rule*: *optimize-primitive-univ.induct*)
(*auto split*: *if-split-asm*)

lemma *optimize-primitive-univ-correct-matchexpr*: **fixes** *m*::*'i*::*len common-primitive match-expr*
shows *matches* (*common-matcher*, *α*) *m* = *matches* (*common-matcher*, *α*) (*optimize-primitive-univ m*)
proof(*simp add*: *fun-eq-iff*, *clarify*, *rename-tac a p*)
fix *a* **and** *p* :: (*'i*::*len*, *'a*) *tagged-packet-scheme*
have *65535* = (−*1*::*16 word*) **by** *simp*
then have *port-range*: ⋀*s e port. s = 0* ∧ *e = 0xFFFF* ⟶ (*port*::*16 word*) ≤ *0xFFFF*
by (*simp only*:) *simp*
have *ternary-ternary-eval* (*map-match-tac common-matcher p m*) = *ternary-ternary-eval*

(*map-match-tac common-matcher p* (*optimize-primitive-univ m*))
apply(*induction m rule*: *optimize-primitive-univ.induct*)
by(*simp-all add*: *port-range match-ifaceAny ipset-from-cidr-0 ctstate-is-UNIV*)

thus *matches* (*common-matcher*, α) *m a p* = *matches* (*common-matcher*, α) (*optimize-primitive-univ m*) *a p*
by(*rule matches-iff-apply-f*)
qed

corollary *optimize-primitive-univ-correct*: *approximating-bigstep-fun* (*common-matcher*, α) *p* (*optimize-matches optimize-primitive-univ rs*) *s* =
approximating-bigstep-fun (*common-matcher*, α) *p rs s*
using *optimize-matches optimize-primitive-univ-correct-matchexpr* **by** *metis*

14.5 Abstracting over unknowns

remove *Extra* (i.e. *TernaryUnknown*) match expressions

fun *upper-closure-matchexpr* :: *action* ⇒ *'i*::*len common-primitive match-expr* ⇒ *'i common-primitive match-expr* **where**
upper-closure-matchexpr - MatchAny = *MatchAny* |
upper-closure-matchexpr Accept (*Match* (*Extra -*)) = *MatchAny* |
upper-closure-matchexpr Reject (*Match* (*Extra -*)) = *MatchNot MatchAny* |
upper-closure-matchexpr Drop (*Match* (*Extra -*)) = *MatchNot MatchAny* |
upper-closure-matchexpr - (*Match m*) = *Match m* |
upper-closure-matchexpr Accept (*MatchNot* (*Match* (*Extra -*))) = *MatchAny* |
upper-closure-matchexpr Drop (*MatchNot* (*Match* (*Extra -*))) = *MatchNot MatchAny* |
upper-closure-matchexpr Reject (*MatchNot* (*Match* (*Extra -*))) = *MatchNot MatchAny* |
upper-closure-matchexpr a (*MatchNot* (*MatchNot m*)) = *upper-closure-matchexpr a m* |
upper-closure-matchexpr a (*MatchNot* (*MatchAnd m1 m2*)) =
(*let m1'* = *upper-closure-matchexpr a* (*MatchNot m1*); *m2'* = *upper-closure-matchexpr a* (*MatchNot m2*) *in*
(*if m1'* = *MatchAny* ∨ *m2'* = *MatchAny*
then MatchAny
else
if m1' = *MatchNot MatchAny then m2' else*
if m2' = *MatchNot MatchAny then m1'*
else
MatchNot (*MatchAnd* (*MatchNot m1'*) (*MatchNot m2'*)))
) |
upper-closure-matchexpr - (*MatchNot m*) = *MatchNot m* |
upper-closure-matchexpr a (*MatchAnd m1 m2*) = *MatchAnd* (*upper-closure-matchexpr a m1*) (*upper-closure-matchexpr a m2*)

lemma *upper-closure-matchexpr-generic*:

$a = Accept \vee a = Drop \Longrightarrow remove\text{-}unknowns\text{-}generic\ (common\text{-}matcher, in\text{-}doubt\text{-}allow)\ a\ m = upper\text{-}closure\text{-}matchexpr\ a\ m$
by(*induction a m rule*: *upper-closure-matchexpr.induct*)
(*simp-all add*: *remove-unknowns-generic-simps2 bool-to-ternary-Unknown common-matcher-SrcDst-defined*)

fun *lower-closure-matchexpr* :: *action* ⇒ *'i::len common-primitive match-expr* ⇒ *'i common-primitive match-expr* **where**
lower-closure-matchexpr - MatchAny = MatchAny |
lower-closure-matchexpr Accept (Match (Extra -)) = MatchNot MatchAny |
lower-closure-matchexpr Reject (Match (Extra -)) = MatchAny |
lower-closure-matchexpr Drop (Match (Extra -)) = MatchAny |
lower-closure-matchexpr - (Match m) = Match m |
lower-closure-matchexpr Accept (MatchNot (Match (Extra -))) = MatchNot MatchAny |
lower-closure-matchexpr Drop (MatchNot (Match (Extra -))) = MatchAny |
lower-closure-matchexpr Reject (MatchNot (Match (Extra -))) = MatchAny |
lower-closure-matchexpr a (MatchNot (MatchNot m)) = lower-closure-matchexpr a m |
lower-closure-matchexpr a (MatchNot (MatchAnd m1 m2)) =
(*let m1' = lower-closure-matchexpr a (MatchNot m1); m2' = lower-closure-matchexpr a (MatchNot m2) in*
(*if m1' = MatchAny ∨ m2' = MatchAny*
then MatchAny
else
if m1' = MatchNot MatchAny then m2' else
if m2' = MatchNot MatchAny then m1'
else
MatchNot (MatchAnd (MatchNot m1') (MatchNot m2'))
) |
lower-closure-matchexpr - (MatchNot m) = MatchNot m |
lower-closure-matchexpr a (MatchAnd m1 m2) = MatchAnd (lower-closure-matchexpr a m1) (lower-closure-matchexpr a m2)

lemma *lower-closure-matchexpr-generic*:
$a = Accept \vee a = Drop \Longrightarrow remove\text{-}unknowns\text{-}generic\ (common\text{-}matcher, in\text{-}doubt\text{-}deny)\ a\ m = lower\text{-}closure\text{-}matchexpr\ a\ m$
by(*induction a m rule*: *lower-closure-matchexpr.induct*)
(*simp-all add*: *remove-unknowns-generic-simps2 bool-to-ternary-Unknown common-matcher-SrcDst-defined*)

end
theory *Example-Semantics*
imports *Call-Return-Unfolding Primitive-Matchers/Common-Primitive-Matcher*
begin

15 Examples Big Step Semantics

We use a primitive matcher which always applies. We don't care about matching in this example.

fun *applies-Yes* :: ('*a*, '*p*) *matcher* **where**
applies-Yes m p = *True*
lemma[*simp*]: *Semantics.matches applies-Yes MatchAny p* **by** *simp*
lemma[*simp*]: *Semantics.matches applies-Yes* (*Match e*) *p* **by** *simp*

definition *m*=*Match* (*Src* (*IpAddr* (*0*::*ipv4addr*)))
lemma[*simp*]: *Semantics.matches applies-Yes m p* **by** (*simp add*: *m-def*)

lemma [''*FORWARD*'' ↦ [(*Rule m Log*), (*Rule m Accept*), (*Rule m Drop*)]],*applies-Yes*,*p*⊢
⟨[*Rule MatchAny* (*Call* ''*FORWARD*'')], *Undecided*⟩ ⇒ (*Decision FinalAllow*)
apply(*rule call-result*)
apply(*auto*)
apply(*rule seq-cons*)
apply(*auto intro*:*Semantics.log*)
apply(*rule seq-cons*)
apply(*auto intro*: *Semantics.accept*)
apply(*rule Semantics.decision*)
done

lemma [''*FORWARD*'' ↦ [(*Rule m Log*), (*Rule m* (*Call* ''*foo*'')), (*Rule m Accept*)],
''*foo*'' ↦ [(*Rule m Log*), (*Rule m Return*)]],*applies-Yes*,*p*⊢
⟨[*Rule MatchAny* (*Call* ''*FORWARD*'')], *Undecided*⟩ ⇒ (*Decision FinalAllow*)
apply(*rule call-result*)
apply(*auto*)
apply(*rule seq-cons*)
apply(*auto intro*: *Semantics.log*)
apply(*rule seq-cons*)
apply(*rule Semantics.call-return*[**where** rs_1=[*Rule m Log*] **and** rs_2=[]])
apply(*simp*)+
apply(*auto intro*: *Semantics.log*)
apply(*auto intro*: *Semantics.accept*)
done

lemma [''*FORWARD*'' ↦ [*Rule m* (*Call* ''*foo*''), *Rule m Drop*], ''*foo*'' ↦ []],*applies-Yes*,*p*⊢
⟨[*Rule MatchAny* (*Call* ''*FORWARD*'')], *Undecided*⟩ ⇒ (*Decision FinalDeny*)
apply(*rule call-result*)
apply(*auto*)
apply(*rule Semantics.seq-cons*)
apply(*rule Semantics.call-result*)
apply(*auto*)
apply(*rule Semantics.skip*)
apply(*auto intro*: *deny*)
done

lemma ((λ*rs*. *process-call* [*''FORWARD''* ↦ [*Rule m* (*Call ''foo''*), *Rule m Drop*], *''foo''* ↦ []] *rs*)^^2)
[*Rule MatchAny* (*Call ''FORWARD''*)]
= [*Rule* (*MatchAnd MatchAny m*) *Drop*] **by** *eval*

hide-const *m*

definition *pkt*=⦇*p-iiface*=*''+''*, *p-oiface*=*''+''*, *p-src*=*0*, *p-dst*=*0*,
p-proto=*TCP*, *p-sport*=*0*, *p-dport*=*0*, *p-tcp-flags* = {*TCP-SYN*},
p-payload=*''''*,*p-tag-ctstate*= *CT-New*⦈

We tune the primitive matcher to support everything we need in the example. Note that the undefined cases cannot be handled with these exact semantics!

fun *applies-exampleMatchExact* :: (*32 common-primitive*, *32 tagged-packet*) *matcher*
where
applies-exampleMatchExact (*Src* (*IpAddr addr*)) *p* ⟷ *p-src p* = *addr* |
applies-exampleMatchExact (*Dst* (*IpAddr addr*)) *p* ⟷ *p-dst p* = *addr* |
applies-exampleMatchExact (*Prot ProtoAny*) *p* ⟷ *True* |
applies-exampleMatchExact (*Prot* (*Proto pr*)) *p* ⟷ *p-proto p* = *pr*

lemma [*''FORWARD''* ↦ [*Rule* (*MatchAnd* (*Match* (*Src* (*IpAddr 0*))) (*Match* (*Dst* (*IpAddr 0*)))) *Reject*,
Rule (*Match* (*Dst* (*IpAddr 0*))) *Log*,
Rule (*Match* (*Prot* (*Proto TCP*))) *Accept*,
Rule (*Match* (*Prot* (*Proto TCP*))) *Drop*]
],*applies-exampleMatchExact*, *pkt*⦇*p-src*:=(*ipv4addr-of-dotdecimal* (*1*,*2*,*3*,*4*)), *p-dst*:=(*ipv4addr-of-dotdecimal* (*0*,*0*,*0*,*0*))⦈⊢
⟨[*Rule MatchAny* (*Call ''FORWARD''*)], *Undecided*⟩ ⇒ (*Decision FinalAllow*)
apply(*rule call-result*)
apply(*auto*)
apply(*rule Semantics.seq-cons*)
apply(*auto intro*: *Semantics.nomatch simp add*: *ipv4addr-of-dotdecimal.simps ipv4addr-of-nat-def*)
apply(*rule Semantics.seq-cons*)
apply(*auto intro*: *Semantics.log simp add*: *ipv4addr-of-dotdecimal.simps ipv4addr-of-nat-def*)
apply(*rule Semantics.seq-cons*)
apply(*auto simp add*: *pkt-def intro*: *Semantics.accept*)
apply(*auto intro*: *Semantics.decision*)
done

end
theory *Alternative-Semantics*
imports *Semantics*
begin

context begin

private inductive *iptables-bigstep-ns* :: *'a ruleset* ⇒ (*'a*, *'p*) *matcher* ⇒ *'p* ⇒ *'a rule list* ⇒ *state* ⇒ *state* ⇒ *bool*
(‹-,-,-⊢ ⟨-, -⟩ ⇒$_s$ -› [*60*,*60*,*60*,*20*,*98*,*98*] *89*)
for Γ **and** γ **and** *p* **where**
skip: Γ,γ,*p*⊢ ⟨[], *t*⟩ ⇒$_s$ *t* |
accept: *matches* γ *m p* ⟹ Γ,γ,*p*⊢ ⟨*Rule m Accept* # *rs*, *Undecided*⟩ ⇒$_s$ *Decision FinalAllow* |
drop: *matches* γ *m p* ⟹ Γ,γ,*p*⊢ ⟨*Rule m Drop* # *rs*, *Undecided*⟩ ⇒$_s$ *Decision FinalDeny* |
reject: *matches* γ *m p* ⟹ Γ,γ,*p*⊢ ⟨*Rule m Reject* # *rs*, *Undecided*⟩ ⇒$_s$ *Decision FinalDeny* |
log: *matches* γ *m p* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_s$ *t* ⟹ Γ,γ,*p*⊢ ⟨*Rule m Log* # *rs*, *Undecided*⟩ ⇒$_s$ *t* |
empty: *matches* γ *m p* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_s$ *t* ⟹ Γ,γ,*p*⊢ ⟨*Rule m Empty* # *rs*, *Undecided*⟩ ⇒$_s$ *t* |
nms: ¬ *matches* γ *m p* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_s$ *t* ⟹ Γ,γ,*p*⊢ ⟨*Rule m a* # *rs*, *Undecided*⟩ ⇒$_s$ *t* |

call-return: ⟦ *matches* γ *m p*; Γ *chain* = *Some* (*rs*$_1$ @ *Rule m'* *Return* # *rs*$_2$);
matches γ *m'* *p*; Γ,γ,*p*⊢ ⟨*rs*$_1$, *Undecided*⟩ ⇒$_s$ *Undecided*; Γ,γ,*p*⊢ ⟨*rrs*, *Undecided*⟩ ⇒$_s$ *t* ⟧ ⟹
Γ,γ,*p*⊢ ⟨*Rule m* (*Call chain*) # *rrs*, *Undecided*⟩ ⇒$_s$ *t* |
call-result: ⟦ *matches* γ *m p*; Γ *chain* = *Some rs*; Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_s$ *Decision X* ⟧ ⟹
Γ,γ,*p*⊢ ⟨*Rule m* (*Call chain*) # *rrs*, *Undecided*⟩ ⇒$_s$ *Decision X* |
call-no-result: ⟦ *matches* γ *m p*; Γ *chain* = *Some rs*; Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_s$ *Undecided*;
Γ,γ,*p*⊢ ⟨*rrs*, *Undecided*⟩ ⇒$_s$ *t* ⟧ ⟹
Γ,γ,*p*⊢ ⟨*Rule m* (*Call chain*) # *rrs*, *Undecided*⟩ ⇒$_s$ *t*

private lemma *a*: Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒$_s$ *t* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t*
apply(*induction rule*: *iptables-bigstep-ns.induct*; (*simp add*: *iptables-bigstep.intros*;*fail*)*?*)
apply (*meson iptables-bigstep.decision iptables-bigstep.accept seq-cons*)
apply (*meson iptables-bigstep.decision iptables-bigstep.drop seq-cons*)
apply (*meson iptables-bigstep.decision iptables-bigstep.reject seq-cons*)
apply (*meson iptables-bigstep.log seq-cons*)
apply (*meson iptables-bigstep.empty seq-cons*)
apply (*meson nomatch seq-cons*)
subgoal using *iptables-bigstep.call-return seq-cons* **by** *fastforce*
apply (*meson iptables-bigstep.decision iptables-bigstep.call-result seq-cons*)
apply (*meson iptables-bigstep.call-result seq'-cons*)
done

private lemma *empty-rs-stateD*: **assumes** Γ,γ,*p*⊢ ⟨[], *s*⟩ ⇒$_s$ *t* **shows** *t* = *s*
using *assms* **by**(*cases rule*: *iptables-bigstep-ns.cases*)
private lemma *decided*: ⟦Γ,γ,*p*⊢ ⟨*rs*$_1$, *Undecided*⟩ ⇒$_s$ *Decision X*⟧ ⟹ Γ,γ,*p*⊢

$\langle rs_1@rs_2,\ Undecided\rangle \Rightarrow_s Decision\ X$
proof(*induction* rs_1)
case *Nil*
then show *?case* **by** (*fast dest*: *empty-rs-stateD*)
next
case (*Cons a* rs_1)
from *Cons.prems* **show** *?case*
by(*cases rule*: *iptables-bigstep-ns.cases*; *simp add*: *Cons.IH iptables-bigstep-ns.intros*)
qed

private lemma *decided-determ*: $[\![\Gamma,\gamma,p\vdash \langle rs_1,\ s\rangle \Rightarrow_s t;\ s = Decision\ X]\!] \Longrightarrow t = Decision\ X$
by(*induction rule*: *iptables-bigstep-ns.induct*; (*simp add*: *iptables-bigstep-ns.intros*;*fail*)*?*)

private lemma *seq-ns*:
$[\![\Gamma,\gamma,p\vdash \langle rs_1,\ Undecided\rangle \Rightarrow_s t;\ \Gamma,\gamma,p\vdash \langle rs_2,\ t\rangle \Rightarrow_s t']\!] \Longrightarrow \Gamma,\gamma,p\vdash \langle rs_1@rs_2, Undecided\rangle \Rightarrow_s t'$
proof (*cases t*, *goal-cases*)
case *1*
from *1*(*1*,*2*) **show** *?case* **unfolding** *1* **proof**(*induction* rs_1)
case (*Cons a* rs_3)
then show *?case*
apply −
apply(*rule iptables-bigstep-ns.cases*[*OF Cons.prems*(*1*)]; *simp add*: *iptables-bigstep-ns.intros*)
done
qed *simp*
next
case (*2 X*)
hence $t' = Decision\ X$ **by** (*simp add*: *decided-determ*)
from *2*(*1*) **show** *?case* **by** (*simp add*: *2*(*3*) ‹$t' = Decision\ X$› *decided*)
qed

private lemma *b*: $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t \Longrightarrow s = Undecided \Longrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_s t$
apply(*induction rule*: *iptables-bigstep.induct*; (*simp add*: *iptables-bigstep-ns.intros*;*fail*)*?*)
apply (*metis decided decision seq-ns seq-progress skipD state.exhaust*)
apply(*metis call-no-result iptables-bigstep-ns.call-result iptables-bigstep-ns.skip state.exhaust*)
done

private inductive *iptables-bigstep-nz* :: *'a ruleset* ⇒ (*'a*, *'p*) *matcher* ⇒ *'p* ⇒ *'a rule list* ⇒ *state* ⇒ *bool*
(‹-,-,-⊢ - ⇒$_z$ ->› [*60*,*60*,*60*,*20*,*98*] *89*)
for Γ **and** γ **and** *p* **where**
skip: $\Gamma,\gamma,p \vdash [] \Rightarrow_z Undecided$ |
accept: *matches* γ *m p* ⟹ $\Gamma,\gamma,p\vdash Rule\ m\ Accept\ \#\ rs \Rightarrow_z Decision\ FinalAllow$ |
drop: *matches* γ *m p* ⟹ $\Gamma,\gamma,p\vdash Rule\ m\ Drop\ \#\ rs \Rightarrow_z Decision\ FinalDeny$ |
reject: *matches* γ *m p* ⟹ $\Gamma,\gamma,p\vdash Rule\ m\ Reject\ \#\ rs \Rightarrow_z Decision\ FinalDeny$ |
log: *matches* γ *m p* ⟹ $\Gamma,\gamma,p\vdash rs \Rightarrow_z t \Longrightarrow \Gamma,\gamma,p\vdash Rule\ m\ Log\ \#\ rs \Rightarrow_z t$ |

empty: *matches* γ *m p* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *rs* $\Rightarrow_z$ *t* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m Empty # rs* $\Rightarrow_z$ *t* |
nms: $\neg$ *matches* γ *m p* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *rs* $\Rightarrow_z$ *t* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m a # rs* $\Rightarrow_z$ *t* |
call-return: $[\![$ *matches* γ *m p*; Γ *chain* = *Some* (*rs*$_1$ @ *Rule m*$'$ *Return # rs*$_2$); *matches* γ *m*$'$ *p*; Γ,γ,*p*$\vdash$ *rs*$_1$ $\Rightarrow_z$ *Undecided*; Γ,γ,*p*$\vdash$ *rrs* $\Rightarrow_z$ *t* $]\!]$ $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m* (*Call chain*) *# rrs* $\Rightarrow_z$ *t* |
call-result: $[\![$ *matches* γ *m p*; Γ *chain* = *Some rs*; Γ,γ,*p*$\vdash$ *rs* $\Rightarrow_z$ *Decision X* $]\!]$ $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m* (*Call chain*) *# rrs* $\Rightarrow_z$ *Decision X* |
call-no-result: $[\![$ *matches* γ *m p*; Γ *chain* = *Some rs*; Γ,γ,*p*$\vdash$ *rs* $\Rightarrow_z$ *Undecided*; Γ,γ,*p*$\vdash$ *rrs* $\Rightarrow_z$ *t* $]\!]$ $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m* (*Call chain*) *# rrs* $\Rightarrow_z$ *t*

private lemma *c*: Γ,γ,*p*$\vdash$ *rs* $\Rightarrow_z$ *t* $\Longrightarrow$ Γ,γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_s$ *t*
by(*induction rule*: *iptables-bigstep-nz.induct*; *simp add*: *iptables-bigstep-ns.intros*)

private lemma *d*: Γ,γ,*p*$\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow_s$ *t* $\Longrightarrow$ *s* = *Undecided* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *rs* $\Rightarrow_z$ *t*
by(*induction rule*: *iptables-bigstep-ns.induct*; *simp add*: *iptables-bigstep-nz.intros*)

inductive *iptables-bigstep-r* :: *'a ruleset* $\Rightarrow$ (*'a*, *'p*) *matcher* $\Rightarrow$ *'p* $\Rightarrow$ *'a rule list* $\Rightarrow$ *state* $\Rightarrow$ *bool*
($\langle$-,-,-$\vdash$ - $\Rightarrow_r$ -$\rangle$ [60,60,60,20,98] 89)
for Γ **and** γ **and** *p* **where**
skip: Γ,γ,*p* $\vdash$ [] $\Rightarrow_r$ *Undecided* |
accept: *matches* γ *m p* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m Accept # rs* $\Rightarrow_r$ *Decision FinalAllow* |
drop: *matches* γ *m p* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m Drop # rs* $\Rightarrow_r$ *Decision FinalDeny* |
reject: *matches* γ *m p* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m Reject # rs* $\Rightarrow_r$ *Decision FinalDeny* |
return: *matches* γ *m p* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m Return # rs* $\Rightarrow_r$ *Undecided* |
log: Γ,γ,*p*$\vdash$ *rs* $\Rightarrow_r$ *t* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m Log # rs* $\Rightarrow_r$ *t* |
empty: Γ,γ,*p*$\vdash$ *rs* $\Rightarrow_r$ *t* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m Empty # rs* $\Rightarrow_r$ *t* |
nms: $\neg$ *matches* γ *m p* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *rs* $\Rightarrow_r$ *t* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m a # rs* $\Rightarrow_r$ *t* |
call-result: $[\![$ *matches* γ *m p*; Γ *chain* = *Some rs*; Γ,γ,*p*$\vdash$ *rs* $\Rightarrow_r$ *Decision X* $]\!]$ $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m* (*Call chain*) *# rrs* $\Rightarrow_r$ *Decision X* |
call-no-result: $[\![$ Γ *chain* = *Some rs*; Γ,γ,*p*$\vdash$ *rs* $\Rightarrow_r$ *Undecided*; Γ,γ,*p*$\vdash$ *rrs* $\Rightarrow_r$ *t* $]\!]$ $\Longrightarrow$ Γ,γ,*p*$\vdash$ *Rule m* (*Call chain*) *# rrs* $\Rightarrow_r$ *t*

private lemma *returning*: $[\![\Gamma,\gamma,p\vdash rs_1 \Rightarrow_r Undecided;$ *matches* γ *m*$'$ *p*$]\!]$ $\Longrightarrow$ Γ,γ,*p*$\vdash$ *rs*$_1$ @ *Rule m*$'$ *Return # rs*$_2$ $\Rightarrow_r$ *Undecided*
proof(*induction rs*$_1$)
case *Nil*
then show *?case* **by** (*simp add*: *return*)
next
case (*Cons a rs*$_3$)
then show *?case* **by** $-$ (*rule iptables-bigstep-r.cases*[*OF Cons.prems*(*1*)]; *simp add*: *iptables-bigstep-r.intros*)
qed

private lemma *e*: Γ,γ,*p*$\vdash$ *rs* $\Rightarrow_z$ *t* $\Longrightarrow$ *s* = *Undecided* $\Longrightarrow$ Γ,γ,*p*$\vdash$ *rs* $\Rightarrow_r$ *t*
by(*induction rule*: *iptables-bigstep-nz.induct*; *simp add*: *iptables-bigstep-r.intros*

returning)

definition *no-call-to c rs* ≡ (∀ *r* ∈ *set rs. case get-action r of Call c'* ⇒ *c* ≠ *c'* | - ⇒ *True*)
definition *all-chains p* Γ *rs* ≡ (*p rs* ∧ (∀ *l rs.* Γ *l* = *Some rs* ⟶ *p rs*))
private lemma *all-chains-no-call-upd*: *all-chains* (*no-call-to c*) Γ *rs* ⟹ (Γ(*c* ↦ *x*)),γ,*p*⊢ *rs* ⇒$_z$ *t* ⟷ Γ,γ,*p*⊢ *rs* ⇒$_z$ *t*
proof (*rule iffI*, *goal-cases*)
case *1*
from *1*(*2*,*1*) **show** *?case*
by(*induction rule*: *iptables-bigstep-nz.induct*;
(*simp add*: *iptables-bigstep-nz.intros no-call-to-def all-chains-def split*: *if-splits*;*fail*) *?*)
next
case *2*
from *2*(*2*,*1*) **show** *?case*
by(*induction rule*: *iptables-bigstep-nz.induct*;
(*simp add*: *iptables-bigstep-nz.intros no-call-to-def all-chains-def split*: *action.splits*;*fail*) *?*)
qed

lemma *updated-call*: Γ(*c* ↦ *rs*),γ,*p*⊢ *rs* ⇒$_z$ *t* ⟹ *matches* γ *m p* ⟹ Γ(*c* ↦ *rs*),γ,*p*⊢ [*Rule m* (*Call c*)] ⇒$_z$ *t*
by(*cases t*; *simp add*: *iptables-bigstep-nz.call-no-result iptables-bigstep-nz.call-result iptables-bigstep-nz.skip*)

private lemma shows
log-nz: Γ,γ,*p*⊢ *rs* ⇒$_z$ *t* ⟹ Γ,γ,*p*⊢ *Rule m Log # rs* ⇒$_z$ *t*
and *empty-nz*: Γ,γ,*p*⊢ *rs* ⇒$_z$ *t* ⟹ Γ,γ,*p*⊢ *Rule m Empty # rs* ⇒$_z$ *t*
by (*meson iptables-bigstep-nz.log iptables-bigstep-nz.empty iptables-bigstep-nz.nms*)+

private lemma *nz-empty-rs-stateD*: **assumes** Γ,γ,*p*⊢ [] ⇒$_z$ *t* **shows** *t* = *Undecided*
using *assms* **by**(*cases rule*: *iptables-bigstep-nz.cases*)

private lemma *upd-callD*: Γ(*c* ↦ *rs*),γ,*p*⊢ [*Rule m* (*Call c*)] ⇒$_z$ *t* ⟹ *matches* γ *m p*
⟹ (Γ(*c* ↦ *rs*),γ,*p*⊢ *rs* ⇒$_z$ *t* ∨ (∃ *rs*$_1$ *rs*$_2$ *m'. rs* = *rs*$_1$ @ *Rule m' Return* # *rs*$_2$ ∧ *matches* γ *m' p* ∧ Γ(*c* ↦ *rs*),γ,*p*⊢ *rs*$_1$ ⇒$_z$ *Undecided* ∧ *t* = *Undecided*))
by(*subst* (*asm*) *iptables-bigstep-nz.simps*) (*auto dest*!: *nz-empty-rs-stateD*)

private lemma *partial-fun-upd*: (*f*(*x* ↦ *y*)) *x* = *Some y* **by**(*fact fun-upd-same*)

lemma *f*: Γ,γ,*p*⊢ *rs* ⇒$_r$ *t* ⟹ *matches* γ *m p* ⟹ *all-chains* (*no-call-to c*) Γ *rs*
⟹
(Γ(*c* ↦ *rs*)),γ,*p*⊢ [*Rule m* (*Call c*)] ⇒$_z$ *t*
proof(*induction rule*: *iptables-bigstep-r.induct*; (*simp add*: *iptables-bigstep-nz.intros*;*fail*) *?*)
case (*return m rs*)
then show *?case* **by** (*metis append-Nil fun-upd-same iptables-bigstep-nz.call-return*

iptables-bigstep-nz.skip)
next
case (*log rs t mx*)
have *ac*: *all-chains* (*no-call-to c*) Γ *rs*
using *log*(*4*) **by**(*simp add*: *all-chains-def no-call-to-def*)
have ∗: Γ(*c* ↦ *Rule mx Log* # *rs*$_1$ @ *Rule m*′ *Return* # *rs*$_2$),γ,*p*⊢ [*Rule m* (*Call c*)] ⇒$_z$ *Undecided*
if *rs* = *rs*$_1$ @ *Rule m*′ *Return* # *rs*$_2$ *matches* γ *m*′ *p*
Γ(*c* ↦ *rs*$_1$ @ *Rule m*′ *Return* # *rs*$_2$),γ,*p*⊢ *rs*$_1$ ⇒$_z$ *Undecided*
for *rs*$_1$ *rs*$_2$ *m*′
proof −
have *ac2*: *all-chains* (*no-call-to c*) Γ *rs*$_1$ **using** *log*(*4*) *that*
by(*simp add*: *all-chains-def no-call-to-def*)
hence Γ(*c* ↦ *Rule mx Log* # *rs*$_1$ @ *Rule m*′ *Return* # *rs*$_2$),γ,*p*⊢ *rs*$_1$ ⇒$_z$ *Undecided*
using *that*(*3*) **unfolding** *that* **by**(*simp add*: *all-chains-no-call-upd*)
hence Γ(*c* ↦ *Rule mx Log* # *rs*$_1$ @ *Rule m*′ *Return* # *rs*$_2$),γ,*p*⊢ *Rule mx Log* # *rs*$_1$ ⇒$_z$ *Undecided*
by (*simp add*: *log-nz*)
thus *?thesis* **using** *that*(*1*,*2*)
by(*elim iptables-bigstep-nz.call-return*[**where** *rs*$_2$=*rs*$_2$, *OF* ‹*matches* γ *m p*›, *rotated*]; *simp add*: *iptables-bigstep-nz.skip*)
qed
from *log*(*2*)[*OF log*(*3*) *ac*] **show** *?case*
apply −
apply(*drule upd-callD*[*OF* - ‹*matches* γ *m p*›])
apply(*erule disjE*)
subgoal
apply(*rule updated-call*[*OF* - ‹*matches* γ *m p*›])
apply(*rule log-nz*)
apply(*simp add*: *ac all-chains-no-call-upd*)
done
using ∗ **by** *blast*
next
case (*empty rs t mx*)

analogous

next
case (*nms m*′ *rs t a*)
have *ac*: *all-chains* (*no-call-to c*) Γ *rs* **using** *nms*(*5*) **by**(*simp add*: *all-chains-def no-call-to-def*)
from *nms.IH*[*OF nms*(*4*) *ac*] **show** *?case*
apply −
apply(*drule upd-callD*[*OF* - ‹*matches* γ *m p*›])
apply(*erule disjE*)
subgoal
apply(*rule updated-call*[*OF* - ‹*matches* γ *m p*›])
apply(*rule iptables-bigstep-nz.nms*[*OF* ‹¬ *matches* γ *m*′ *p*›])
apply(*simp add*: *ac all-chains-no-call-upd*)

```
      done
    apply safe
    subgoal for rs1 rs2 r
     apply(subgoal-tac all-chains (no-call-to c) Γ rs1)
      apply(subst (asm) all-chains-no-call-upd, assumption)
      apply(subst (asm) all-chains-no-call-upd[symmetric], assumption)
      apply(drule iptables-bigstep-nz.nms[where a=a, OF ‹¬ matches γ m' p›])
     apply(erule (1) iptables-bigstep-nz.call-return[where rs2=rs2, OF ‹matches
γ m p›, rotated])
     apply(insert ac; simp add: all-chains-def no-call-to-def iptables-bigstep-nz.skip)+
      done
    done
next
  case (call-result m' c' rs X rrs)
   have acrs: all-chains (no-call-to c) Γ rs using call-result(2,6) by(simp add:
all-chains-def no-call-to-def)
  have cc: c ≠ c' using call-result(6) by(simp add: all-chains-def no-call-to-def)
   have Γ(c ↦ rs),γ,p⊢ [Rule m (Call c)] ⇒z Decision X using call-result.IH
call-result.prems(1) acrs by blast
  then show ?case
    apply −
    apply(drule upd-callD[OF - ‹matches γ m p›])
    apply(erule disjE)
    subgoal
     apply(rule updated-call[OF - ‹matches γ m p›])
     apply(rule iptables-bigstep-nz.call-result[where rs=rs, OF ‹matches γ m' p›
])
     apply(simp add: cc[symmetric] call-result(2);fail)
     apply(simp add: acrs all-chains-no-call-upd;fail)
     done
    apply safe
  done
next
  case (call-no-result c' rs rrs t m')
  have acrs: all-chains (no-call-to c) Γ rs using call-no-result(1,7) by(simp add:
all-chains-def no-call-to-def)
  have acrrs: all-chains (no-call-to c) Γ rrs using call-no-result(7) by(simp add:
all-chains-def no-call-to-def)
  have acrs1: all-chains (no-call-to c) Γ rs1 if rs = rs1 @ rs2 for rs1 rs2
    using acrs that by(simp add: all-chains-def no-call-to-def)
  have acrrs1: all-chains (no-call-to c) Γ rs1 if rrs = rs1 @ rs2 for rs1 rs2
    using acrrs that by(simp add: all-chains-def no-call-to-def)
 have cc: c ≠ c' using call-no-result(7) by(simp add: all-chains-def no-call-to-def)
  have ∗: Γ(c ↦ rs),γ,p⊢ [Rule m (Call c)] ⇒z Undecided using call-no-result.IH
call-no-result.prems(1) acrs by blast
 have ∗∗: Γ(c ↦ rrs),γ,p⊢ [Rule m (Call c)] ⇒z t by (simp add: acrrs call-no-result.IH(2)
call-no-result.prems(1))
  show ?case proof(cases ‹matches γ m' p›)
    case True
```

from *call-no-result*(*5*)[*OF* ‹*matches* γ *m p*› *acrrs*] ∗ **show** *?thesis*
apply −
apply(*drule upd-callD*[*OF* - ‹*matches* γ *m p*›])+
apply(*elim disjE*)
apply *safe*
subgoal
apply(*rule updated-call*[*OF* - ‹*matches* γ *m p*›])
apply(*rule iptables-bigstep-nz.call-no-result*[**where** *rs*=*rs*, *OF* ‹*matches* γ m' *p*›])
apply(*simp add*: *cc*[*symmetric*] *call-no-result*(*1*);*fail*)
apply(*simp add*: *acrs all-chains-no-call-upd*;*fail*)
apply(*simp add*: *acrrs all-chains-no-call-upd*)
done
subgoal for rs_1 rs_2 *r*
apply(*rule updated-call*[*OF* - ‹*matches* γ *m p*›])
apply(*rule call-return*[*OF* ‹*matches* γ m' *p*›])
apply(*simp add*: *cc*[*symmetric*] *call-no-result*(*1*);*fail*)
apply(*simp*;*fail*)
apply(*simp add*: *acrs1 all-chains-no-call-upd*;*fail*)
apply(*simp add*: *acrrs all-chains-no-call-upd*)
done
subgoal for rs_1 rs_2 *r*
apply(*rule call-return*[**where** rs_1=*Rule* m' (*Call* c') # rs_1, *OF* ‹*matches* γ *m p*›])
apply(*simp*;*fail*)
apply(*simp*;*fail*)
apply(*rule iptables-bigstep-nz.call-no-result*[*OF* ‹*matches* γ m' *p*›])
apply(*simp add*: *cc*[*symmetric*] *call-no-result*(*1*);*fail*)
apply (*meson acrs all-chains-no-call-upd*)
apply(*subst all-chains-no-call-upd*; *simp add*: *acrrs1 all-chains-no-call-upd*; *fail*)
apply (*simp add*: *iptables-bigstep-nz.skip*;*fail*)
done
subgoal for rrs_1 rs_1 rrs_2 rs_2 *rr r*
apply(*rule call-return*[**where** rs_1=*Rule* m' (*Call* c') # rrs_1, *OF* ‹*matches* γ *m p*›])
apply(*simp*;*fail*)
apply(*simp*;*fail*)
apply(*rule call-return*[*OF* ‹*matches* γ m' *p*›])
apply(*simp add*: *cc*[*symmetric*] *call-no-result*(*1*);*fail*)
apply *blast*
apply (*meson acrs1 all-chains-no-call-upd*)
apply(*subst all-chains-no-call-upd*; *simp add*: *acrrs1 all-chains-no-call-upd*; *fail*)
apply (*simp add*: *iptables-bigstep-nz.skip*;*fail*)
done
done
next
case *False*

```
    from iptables-bigstep-nz.nms[OF False] ** show ?thesis
      apply -
      apply(drule upd-callD[OF - ‹matches γ m p›])+
      apply(elim disjE)
      subgoal
        apply(rule updated-call[OF - ‹matches γ m p›])
        apply(rule iptables-bigstep-nz.nms[OF False])
        apply(simp add: acrrs all-chains-no-call-upd)
        done
      apply safe
      subgoal for rs1 rs2 r
       apply(rule call-return[where rs1=Rule m' (Call c') # rs1, OF ‹matches γ
m p›])
          apply(simp;fail)
         apply(simp;fail)
        apply(rule iptables-bigstep-nz.nms[OF False])
       apply(subst all-chains-no-call-upd; simp add: acrrs1 all-chains-no-call-upd;
fail)
       apply(simp add: iptables-bigstep-nz.skip;fail)
       done
     done
  qed
qed

lemma r-skip-inv: Γ,γ,p⊢ [] ⇒r t ⟹ t = Undecided
  by(subst (asm) iptables-bigstep-r.simps) auto


lemma r-call-eq: Γ c = Some rs ⟹ matches γ m p ⟹ Γ,γ,p⊢ [Rule m (Call c)]
⇒r t ⟷ Γ,γ,p⊢ rs ⇒r t

  apply(rule iffI)
  subgoal
    apply(subst (asm) iptables-bigstep-r.simps)
    apply(auto dest: r-skip-inv)
  done
  subgoal
    apply(cases t)
     apply(erule iptables-bigstep-r.call-no-result)
      apply(simp;fail)
     apply(simp add: iptables-bigstep-r.skip;fail)
      apply(simp)
    apply(erule (2) iptables-bigstep-r.call-result)
  done
  by -


lemma call-eq: Γ c = Some rs ⟹ matches γ m p ⟹ ∀r ∈ set rs. get-action r
≠ Return ⟹ Γ,γ,p⊢ ⟨[Rule m (Call c)],s⟩ ⇒ t ⟷ Γ,γ,p⊢ ⟨rs,s⟩ ⇒ t
```

apply(*rule iffI*)
subgoal
apply(*subst* (*asm*) *iptables-bigstep.simps*)
apply (*auto*)
apply (*simp add*: *decision*)
apply(*erule rules-singleton-rev-E*; *simp*; *metis callD in-set-conv-decomp rule.sel*(*2*) *skipD*)
done
by (*metis decision iptables-bigstep.call-result iptables-bigstep-deterministic state.exhaust*)

theorem *r-eq-orig*: ⟦*all-chains* (*no-call-to c*) Γ *rs*; Γ *c* = *Some rs*⟧ ⟹
Γ,γ,*p*⊢ *rs* ⇒$_r$ *t* ⟷ Γ,γ,*p*⊢ ⟨[*Rule MatchAny* (*Call c*)], *Undecided*⟩ ⇒ *t*
apply(*rule iffI*)
subgoal
apply(*drule f*[**where** *m*=*MatchAny*, *THEN c*, *THEN a*])
apply(*simp*;*fail*)
apply(*simp*;*fail*)
apply (*metis fun-upd-triv*)
done
subgoal
apply(*subst r-call-eq*[**where** *m*=*MatchAny*, *symmetric*])
apply(*simp*;*fail*)
apply(*simp*;*fail*)
apply(*erule b*[*THEN d*, *THEN e*, *OF - refl refl refl*])
done
done

lemma *r-no-call*: Γ,γ,*p*⊢ *Rule MatchAny* (*Call c*)#*rs* ⇒$_r$ *t* ⟹ Γ *c* = *None* ⟹ *False*
by(*subst* (*asm*) *iptables-bigstep-r.simps*) *simp*

lemma *no-call*: Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t* ⟹ *rs* = [*Rule MatchAny* (*Call c*)] ⟹ *s* = *Undecided* ⟹ Γ *c* = *None* ⟹ *False*
by (*meson b d e r-no-call*)

private corollary *r-eq-orig′*: **assumes** ∀ *rs* ∈ *ran* Γ. *no-call-to c rs*
shows Γ,γ,*p*⊢ [*Rule MatchAny* (*Call c*)] ⇒$_r$ *t* ⟷ Γ,γ,*p*⊢ ⟨[*Rule MatchAny* (*Call c*)], *Undecided*⟩ ⇒ *t*

proof −
show *?thesis* **proof** (*cases* Γ *c*)
fix *rs*
assume Γ *c* = *Some rs*
moreover hence *all-chains* (*no-call-to c*) Γ *rs* **using** *assms* **by** (*simp add*: *all-chains-def ranI*)
ultimately show *?thesis* **by**(*simp add*: *r-call-eq r-eq-orig*)
next
assume Γ *c* = *None* **thus** *?thesis* **using** *r-no-call no-call* **by** *metis*

```
  qed
qed


lemma r-tail: assumes Γ,γ,p⊢ rs1 ⇒_r Decision X shows Γ,γ,p⊢ rs1 @ rs2 ⇒_r
Decision X
proof −
  have Γ,γ,p⊢ rs1 ⇒_r t ⟹ t = Decision X ⟹ Γ,γ,p⊢ rs1 @ rs2 ⇒_r Decision
X for t
    by(induction rule: iptables-bigstep-r.induct; simp add: iptables-bigstep-r.intros)
  thus ?thesis using assms by blast
qed
lemma r-seq: Γ,γ,p⊢ rs1 ⇒_r Undecided ⟹ ∀ r ∈ set rs1. ¬(get-action r = Return
∧ matches γ (get-match r) p)
   ⟹ Γ,γ,p⊢ rs2 ⇒_r t ⟹ Γ,γ,p⊢ rs1 @ rs2 ⇒_r t
proof(induction rs1)
  case Nil
  then show ?case by simp
next
  case (Cons r rs1)
  have p2: ∀ r∈set rs1. ¬ (get-action r = Return ∧ matches γ (get-match r) p)
          ¬(get-action r = Return ∧ matches γ (get-match r) p)
    by (simp-all add: Cons.prems(2))
  from Cons.prems(1) p2(2) Cons.IH[OF - p2(1) Cons.prems(3)] show ?case
    by(cases rule: iptables-bigstep-r.cases; simp add: iptables-bigstep-r.intros)
qed

lemma r-appendD: Γ,γ,p⊢ rs1 @ rs2 ⇒_r t ⟹ ∃ s. Γ,γ,p⊢ rs1 ⇒_r s
  proof(induction rs1)
    case (Cons r rs1)
    from Cons.prems Cons.IH show ?case by(cases rule: iptables-bigstep-r.cases)
(auto intro: iptables-bigstep-r.intros)
  qed (meson iptables-bigstep-r.skip)

corollary iptables-bigstep-r-eq: assumes ∀ rs ∈ ran Γ. no-call-to c rs A = Accept
∨ A = Drop
  shows Γ,γ,p⊢ [Rule MatchAny (Call c), Rule MatchAny A] ⇒_r t ⟷ Γ,γ,p⊢
⟨[Rule MatchAny (Call c), Rule MatchAny A], Undecided⟩ ⇒ t

proof −
  show ?thesis proof (cases Γ c)
    fix rs
    assume Γ c = Some rs
     moreover hence all-chains (no-call-to c) Γ rs using assms by (simp add:
all-chains-def ranI)
    show ?thesis

      apply(rule iffI[rotated])
       apply(erule seqE-cons)
```

apply(*subst* (*asm*) *r-eq-orig'*[*symmetric*])
apply (*simp add*: *assms*(*1*);*fail*)
apply (*meson assms*(*1*) *b d e r-eq-orig' seq'-cons*)
apply(*frule r-appendD*[*of - - -* [*Rule MatchAny* (*Call c*)] [*Rule MatchAny A*], *simplified*])
apply(*subst* (*asm*) *r-eq-orig'*)
apply (*simp add*: *assms*(*1*);*fail*)
apply(*clarsimp*)
apply(*subst* (*asm*) *r-eq-orig'*[*symmetric*])
apply (*simp add*: *assms*(*1*);*fail*)
apply(*subst* (*asm*)(*2*) *iptables-bigstep-r.simps*)
apply(*subst* (*asm*)(*1*) *iptables-bigstep-r.simps*)
apply *auto*
apply (*metis append-Cons append-Nil assms*(*1*) *decision matches.simps*(*4*) *r-call-eq r-eq-orig' seq*)
apply (*metis* ‹*all-chains* (*no-call-to c*) Γ *rs*› *calculation iptables-bigstep-deterministic option.inject r-eq-orig state.distinct*(*1*))
subgoal using ‹*all-chains* (*no-call-to c*) Γ *rs*› *calculation iptables-bigstep-deterministic r-eq-orig* **by** *fastforce*
apply(*subst* (*asm*) *r-eq-orig*[*rotated*])
apply(*assumption*)
subgoal using ‹*all-chains* (*no-call-to c*) Γ *rs*› *calculation* **by** *simp*
apply(*erule seq'-cons*)
apply(*subst* (*asm*)(*1*) *iptables-bigstep-r.simps*)
apply(*insert assms*(*2*); *auto simp add*: *iptables-bigstep.intros*)
done
next
assume Γ *c* = *None* **thus** *?thesis* **using** *r-no-call no-call* **by** (*metis seqE-cons*)
qed
qed

lemma *ex-no-call*: *finite S* ⟹ ∃ *c*. ∀ (*rs* :: *'a rule list*) ∈ *S*. *no-call-to c rs*

proof −
assume *fS*: ‹*finite S*›
define *called-c* **where** *called-c rs* = {*c*. ∃ *m*. *Rule m* (*Call c*) ∈ *set rs*} **for** *rs* :: *'a rule list*
define *called-c'* **where** *called-c' rs* = *set* [*c*. *r* ← *rs*, *c* ← (*case get-action r of Call c* ⇒ [*c*] | - ⇒ [])]
for *rs* :: *'a rule list*
have *cc*: *called-c' rs* = *called-c rs* **for** *rs*
unfolding *called-c'-def called-c-def*
by(*induction rs*; *simp add*: *Un-def*) (*auto*; *metis rule.collapse*)
have *f*: *finite* (*called-c rs*) **for** *rs* **unfolding** *cc*[*symmetric*] *called-c'-def* **by** *blast*
have *ncc*: *no-call-to c rs* ⟷ *c* ∉ *called-c rs* **for** *c rs*
by(*induction rs*; *auto simp add*: *no-call-to-def called-c-def split*: *action.splits*) (*metis rule.collapse*)
have *isu*: *infinite* (*UNIV* :: *string set*) **by** (*simp add*: *infinite-UNIV-listI*)

```
  have ff: finite (⋃ rs ∈ S. called-c rs) using f fS by simp
  then obtain c where ne: c ∉ (⋃ rs ∈ S. called-c rs)
    by (blast dest: ex-new-if-finite[OF isu])
  thus ?thesis by(intro exI[where x=c]) (simp add: ncc)

qed

private lemma ex-no-call': finite (dom Γ) ⟹ ∃ c. Γ c = None ∧ (∀ (rs :: 'a rule
list) ∈ (ran Γ). no-call-to c rs)

proof −
  have ∗: finite S ⟹ (dom M) = S ⟹ ∃ m. M = map-of m for M S
  proof(induction arbitrary: M rule: finite.induct)
    case emptyI
    then show ?case by(intro exI[where x=Nil]) simp
  next
    case (insertI A a)
    show ?case proof(cases a ∈ A)
      case True
      then show ?thesis using insertI by (simp add: insert-absorb)
    next
      case False
      hence dom (M(a := None)) = A using insertI.prems by simp
      from insertI.IH[OF this] obtain m where M(a := None) = map-of m ..
      then show ?thesis
      by(intro exI[where x=(a, the (M a)) # m]) (simp; metis domIff fun-upd-apply
insertCI insertI.prems option.collapse)
    qed
  qed
  have ran-alt: ran f = (the o f) ‘ dom f for f by(auto simp add: ran-def dom-def
image-def)
  assume fD: ‹finite (dom Γ)›
  hence fS: ‹finite (ran Γ)› by(simp add: ran-alt)
  define called-c where called-c rs = {c. ∃ m. Rule m (Call c) ∈ set rs} for rs ::
'a rule list
  define called-c' where called-c' rs = set [c. r ← rs, c ← (case get-action r of
Call c ⇒ [c] | - ⇒ [])]
    for rs :: 'a rule list
  have cc: called-c' rs = called-c rs for rs
    unfolding called-c'-def called-c-def
    by(induction rs; simp add: Un-def) (auto; metis rule.collapse)
  have f: finite (called-c rs) for rs unfolding cc[symmetric] called-c'-def by blast
  have ncc: no-call-to c rs ⟷ c ∉ called-c rs for c rs
    by(induction rs; auto simp add: no-call-to-def called-c-def split: action.splits)
(metis rule.collapse)
  have isu: infinite (UNIV :: string set) by (simp add: infinite-UNIV-listI)
  have ff: finite (⋃ rs ∈ ran Γ. called-c rs) using f fS by simp
  hence fff: finite (dom Γ ∪ (⋃ rs ∈ ran Γ. called-c rs)) using fD by simp
   then obtain c where ne: c ∉ (dom Γ ∪ (⋃ rs ∈ ran Γ. called-c rs)) thm
```

ex-new-if-finite
by (*metis UNIV-I isu set-eqI*)
thus *?thesis* **by**(*fastforce simp add*: *ncc*)
qed

lemma *all-chains-no-call-upd-r*: *all-chains* (*no-call-to c*) Γ *rs* ⟹ (Γ(*c* ↦ *x*)),γ,*p*⊢ *rs* ⇒$_r$ *t* ⟷ Γ,γ,*p*⊢ *rs* ⇒$_r$ *t*
proof (*rule iffI*, *goal-cases*)
case *1*
from *1*(*2*,*1*) **show** *?case*
by(*induction rule*: *iptables-bigstep-r.induct*;
(*simp add*: *iptables-bigstep-r.intros no-call-to-def all-chains-def split*: *if-splits*;*fail*) *?*)
next
case *2*
from *2*(*2*,*1*) **show** *?case*
by(*induction rule*: *iptables-bigstep-r.induct*;
(*simp add*: *iptables-bigstep-r.intros no-call-to-def all-chains-def split*: *action.splits*;*fail*) *?*)
qed

lemma *all-chains-no-call-upd-orig*: *all-chains* (*no-call-to c*) Γ *rs* ⟹ (Γ(*c* ↦ *x*)),γ,*p*⊢ ⟨*rs*,*s*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨*rs*,*s*⟩ ⇒ *t*
proof (*rule iffI*, *goal-cases*)
case *1*
from *1*(*2*,*1*) **show** *?case*
by(*induction rs s t rule*: *iptables-bigstep.induct*;
(*simp add*: *iptables-bigstep.intros no-call-to-def all-chains-def split*: *if-splits*;*fail*) *?*)
next
case *2*
from *2*(*2*,*1*) **show** *?case*
by(*induction rule*: *iptables-bigstep.induct*;
(*simp add*: *iptables-bigstep.intros no-call-to-def all-chains-def split*: *action.splits*;*fail*) *?*)
qed

corollary *r-eq-orig'''*: **assumes** *finite* (*ran* Γ) **and** ∀ *r* ∈ *set rs*. *get-action r* ≠ *Return*
shows Γ,γ,*p*⊢ *rs* ⇒$_r$ *t* ⟷ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *t*
proof −
from *assms* **have** *finite* ({*rs*} ∪ (*ran* Γ)) **by** *simp*
from *ex-no-call*[*OF this*] **obtain** *c* **where** *c*: (∀ *rs*∈*ran* Γ. *no-call-to c rs*) *no-call-to c rs* **by** *blast*
hence *acnc*: *all-chains* (*no-call-to c*) Γ *rs* **unfolding** *all-chains-def* **by** (*simp add*: *ranI*)
have *ranaway*: ∀ *rs*∈*ran* (Γ(*c* ↦ *rs*)). *no-call-to c rs*
proof −
{
fix *rsa* :: *'a rule list*

assume *a1*: *rsa* ∈ *ran* (Γ(*c* ↦ *rs*))
have ⋀*R*. *rs* ∈ *R* ∪ *Collect* (*no-call-to c*)
using *c*(*2*) **by** *force*
then have *rsa* ∈ *ran* (Γ(*c* := *None*)) ∪ *Collect* (*no-call-to c*)
using *a1* **by** (*metis* (*no-types*) *Un-iff Un-insert-left fun-upd-same fun-upd-upd insert-absorb ran-map-upd*)
then have *no-call-to c rsa*
by (*metis* (*no-types*) *Un-iff c*(*1*) *mem-Collect-eq ranI ran-restrictD restrict-complement-singleton-eq*)
}
thus *?thesis* **by** *simp*
qed
have Γ(*c* ↦ *rs*),γ,*p*⊢ *rs* ⇒*r* *t* ⟷ Γ(*c* ↦ *rs*),γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *t*
apply(*subst r-call-eq*[**where** *c*=*c* **and** *m*=*MatchAny*,*symmetric*])
apply(*simp*;*fail*)
apply(*simp*;*fail*)
apply(*subst call-eq*[**where** *c*=*c* **and** *m*=*MatchAny*,*symmetric*])
apply(*simp*;*fail*)
apply(*simp*;*fail*)
apply(*simp add*: *assms*;*fail*)
apply(*rule r-eq-orig'*)
apply(*fact ranaway*)
done
thus *?thesis*
apply −
apply(*subst* (*asm*) *all-chains-no-call-upd-r*[**where** *x*=*rs*, *OF acnc*])
apply(*subst* (*asm*) *all-chains-no-call-upd-orig*[**where** *x*=*rs*, *OF acnc*])
.
qed

end

end
theory *Semantics-Stateful*
imports *Semantics*
begin

16 Semantics Stateful

16.1 Model 1 – Curried Stateful Matcher

Processing a packet with state can be modeled as follows: The state is σ. The primitive matcher γ_σ is a curried function where the first argument is the state and it returns a stateless primitive matcher, i.e. $\gamma = \gamma_\sigma\ \sigma$. With this stateless primitive matcher γ, the *iptables-bigstep* semantics are executed. As entry point, the iptables built-in chains *"INPUT"*, *"OUTPUT"*, and *"FORWARD"* with their default-policy (*Accept* or *Drop* are valid for iptables) are chosen. The semantics must yield a *Decision X*. Due to the

default-policy, this is always the case if the ruleset is well-formed. When a decision is made, the state σ is updated.

inductive *semantics-stateful* ::
$'a$ *ruleset* $\Rightarrow$
($'\sigma \Rightarrow$ ($'a$, $'p$) *matcher*) $\Rightarrow$ — matcher, first parameter is the state
($'\sigma \Rightarrow$ *final-decision* $\Rightarrow$ $'p \Rightarrow$ $'\sigma$) $\Rightarrow$ — state update function after firewall has decision for a packet
$'\sigma \Rightarrow$ — Starting state. constant
(*string* $\times$ *action*) $\Rightarrow$ — The chain and default policy the firewall evaluates. For example "FORWARD", Drop
$'p$ *list* $\Rightarrow$ — packets to be processed
($'p$ $\times$ *final-decision*) *list* $\Rightarrow$ — packets which have been processed and their decision. ordered the same as the firewall processed them. oldest packet first
$'\sigma \Rightarrow$ — final state
bool **for** Γ **and** γ_σ **and** *state-update* **and** σ_0 **where**
— A list of packets *ps* waiting to be processed. Nothing has happened, start and final state are the same, the list of processed packets is empty.
semantics-stateful Γ γ_σ *state-update* σ_0 (*built-in-chain*, *default-policy*) *ps* [] σ_0 |

— Processing one packet
semantics-stateful Γ γ_σ *state-update* σ_0 (*built-in-chain*, *default-policy*) (*p*#*ps*) *ps-processed* σ' $\Longrightarrow$
Γ,(γ_σ σ'),*p*$\vdash$ $\langle$[*Rule MatchAny* (*Call built-in-chain*), *Rule MatchAny default-policy*],*Undecided*$\rangle$ $\Rightarrow$ *Decision X* $\Longrightarrow$
semantics-stateful Γ γ_σ *state-update* σ_0 (*built-in-chain*, *default-policy*) *ps* (*ps-processed*@[(*p*, *X*)]) (*state-update* σ' *X p*)

lemma *semantics-stateful-intro-process-one*: *semantics-stateful* Γ γ_σ *state-upate* σ_0 (*built-in-chain*, *default-policy*) (*p*#*ps*) *ps-processed-old* σ*-old* $\Longrightarrow$
Γ,γ_σ σ*-old*,*p*$\vdash$ $\langle$[*Rule MatchAny* (*Call built-in-chain*), *Rule MatchAny default-policy*], *Undecided*$\rangle$ $\Rightarrow$ *Decision X* $\Longrightarrow$
σ' = *state-upate* σ*-old X p* $\Longrightarrow$
ps-processed = *ps-processed-old*@[(*p*, *X*)] $\Longrightarrow$
semantics-stateful Γ γ_σ *state-upate* σ_0 (*built-in-chain*, *default-policy*) *ps ps-processed* σ'
by(*auto intro*: *semantics-stateful.intros*)

lemma *semantics-stateful-intro-start*: $\sigma_0 = \sigma' \Longrightarrow$ *ps-processed* = [] $\Longrightarrow$
semantics-stateful Γ γ_σ *state-upate* σ_0 (*built-in-chain*, *default-policy*) *ps ps-processed* σ'
by(*auto intro*: *semantics-stateful.intros*)

Example below

16.2 Model 2 – Packets Tagged with State Information

In this model, the matcher is completely stateless but packets are previously tagged with (static) stateful information.

inductive *semantics-stateful-packet-tagging* ::
'a ruleset ⇒
(*'a*, *'ptagged*) *matcher* ⇒
(*'σ* ⇒ *'p* ⇒ *'ptagged*) ⇒ — taggs the packet accordig to the current state before processing by firewall
(*'σ* ⇒ *final-decision* ⇒ *'p* ⇒ *'σ*) ⇒ — state updater
'σ ⇒ — Starting state. constant
(*string* × *action*) ⇒
'p list ⇒ — packets to be processed
(*'p* × *final-decision*) *list* ⇒ — packets which have been processed
'σ ⇒ — final state
bool **for** Γ **and** *γ* **and** *packet-tagger* **and** *state-update* **and** σ_0 **where**
semantics-stateful-packet-tagging Γ *γ packet-tagger state-update* σ_0 (*built-in-chain*, *default-policy*) *ps* [] σ_0 |

semantics-stateful-packet-tagging Γ *γ packet-tagger state-update* σ_0 (*built-in-chain*, *default-policy*) (*p*#*ps*) *ps-processed σ'* ⟹
Γ,*γ*,(*packet-tagger σ' p*)⊢ ⟨[*Rule MatchAny* (*Call built-in-chain*), *Rule MatchAny default-policy*],*Undecided*⟩ ⇒ *Decision X* ⟹
semantics-stateful-packet-tagging Γ *γ packet-tagger state-update* σ_0 (*built-in-chain*, *default-policy*) *ps* (*ps-processed*@[(*p*, *X*)]) (*state-update σ' X p*)

lemma *semantics-stateful-packet-tagging-intro-start*: $\sigma_0 = \sigma' \Longrightarrow$ *ps-processed* = [] ⟹
semantics-stateful-packet-tagging Γ *γ packet-tagger state-upate* σ_0 (*built-in-chain*, *default-policy*) *ps ps-processed σ'*
by(*auto intro*: *semantics-stateful-packet-tagging.intros*)

lemma *semantics-stateful-packet-tagging-intro-process-one*:
semantics-stateful-packet-tagging Γ *γ packet-tagger state-upate* σ_0 (*built-in-chain*, *default-policy*) (*p*#*ps*) *ps-processed-old σ-old* ⟹
Γ,*γ*,(*packet-tagger σ-old p*)⊢ ⟨[*Rule MatchAny* (*Call built-in-chain*), *Rule MatchAny default-policy*], *Undecided*⟩ ⇒ *Decision X* ⟹
σ' = *state-upate σ-old X p* ⟹
ps-processed = *ps-processed-old*@[(*p*, *X*)] ⟹
semantics-stateful-packet-tagging Γ *γ packet-tagger state-upate* σ_0 (*built-in-chain*, *default-policy*) *ps ps-processed σ'*
by(*auto intro*: *semantics-stateful-packet-tagging.intros*)

lemma *semantics-bigstep-state-vs-tagged*:
assumes ∀ *m*::*'m*. *stateful-matcher' σ m p* = *stateful-matcher-tagged' m* (*packet-tagger' σ p*)
shows Γ,*stateful-matcher' σ*,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *t* ⟷
Γ,*stateful-matcher-tagged'*,*packet-tagger' σ p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *t*
proof −
{ fix *m*::*'m match-expr*
from *assms* **have**

```
    matches (stateful-matcher' σ) m p ⟷ matches stateful-matcher-tagged' m
(packet-tagger' σ p)
     by(induction m) (simp-all)
  } note matches-stateful-matcher-stateful-matcher-tagged=this

  show ?thesis (is ?lhs ⟷ ?rhs)
  proof
    assume ?lhs
    thus ?rhs
     proof(induction rs Undecided t rule: iptables-bigstep-induct)
     case (Seq - - - t)
       thus ?case
         apply(cases t)
          apply (simp add: seq)
         apply(auto simp add: decision seq dest: decisionD)
         done
   qed(auto intro: iptables-bigstep.intros simp add: matches-stateful-matcher-stateful-matcher-tagged)
  next
    assume ?rhs
    thus ?lhs
     proof(induction rs Undecided t rule: iptables-bigstep-induct)
     case (Seq - - - t)
       thus ?case
         apply(cases t)
          apply (simp add: seq)
         apply(auto  simp add: decision seq dest: decisionD)
         done
   qed(auto intro: iptables-bigstep.intros simp add: matches-stateful-matcher-stateful-matcher-tagged)
  qed
qed
```

Both semantics are equal

```
theorem semantics-stateful-vs-tagged:
  assumes ∀ m σ p. stateful-matcher' σ m p = stateful-matcher-tagged' m (packet-tagger'
σ p)
  shows semantics-stateful rs stateful-matcher' state-update' σ₀ start ps ps-processed
σ' =
        semantics-stateful-packet-tagging rs stateful-matcher-tagged' packet-tagger'
state-update' σ₀ start ps ps-processed σ'
  proof −
  from semantics-bigstep-state-vs-tagged[of stateful-matcher' - - stateful-matcher-tagged'
packet-tagger'] assms
     have vs-tagged:
    rs,stateful-matcher' σ',p⊢ ⟨[Rule MatchAny (Call built-in-chain), Rule MatchAny
default-policy], Undecided⟩ ⇒ t ⟷
     rs,stateful-matcher-tagged',packet-tagger' σ' p⊢ ⟨[Rule MatchAny (Call built-in-chain),
Rule MatchAny default-policy], Undecided⟩ ⇒ t
      for t p σ' built-in-chain default-policy by blast
   from assms have stateful-matcher-eq:
```

```
    (λa b. stateful-matcher-tagged' a (packet-tagger' σ' b)) = stateful-matcher' σ'
for σ' by presburger
  show ?thesis (is ?lhs ⟷ ?rhs)
    proof
      assume ?lhs thus ?rhs
        proof(induction rule: semantics-stateful.induct)
      case 1 thus ?case by(auto intro: semantics-stateful-packet-tagging-intro-start)[1]
        next
        case (2 built-in-chain default-policy p ps  ps-processed σ')
          from 2 have
          semantics-stateful-packet-tagging rs stateful-matcher-tagged' packet-tagger'
state-update' σ₀ (built-in-chain, default-policy) (p # ps) ps-processed σ'
            by blast
          with 2(2,3) show ?case
          apply −
          apply(rule semantics-stateful-packet-tagging-intro-process-one)
            apply(simp-all add: vs-tagged)
          done
       qed
    next
      assume ?rhs thus ?lhs
      proof(induction rule: semantics-stateful-packet-tagging.induct)
        case 1 thus ?case by(auto intro: semantics-stateful-intro-start)
        next
        case (2 built-in-chain default-policy p ps ps-processed σ') thus ?case
          apply −
          apply(rule semantics-stateful-intro-process-one)
            apply(simp-all add: stateful-matcher-eq vs-tagged)
          done
      qed
    qed
  qed
```

Examples

```
context
begin
```

16.3 Example: Conntrack with curried matcher

We illustrate stateful semantics with a simple example. We allow matching on the states New and Established. In addition, we introduce a primitive match to match on outgoing ssh packets (dst port = 22). The state is managed in a state table where accepted connections are remembered.

SomePacket with source and destination port or something we don't know about

```
  private datatype packet = SomePacket nat × nat | OtherPacket
```

private datatype *primitive-matches* = *New* | *Established* | *IsSSH*

In the state, we remember the packets which belong to an established connection.

private datatype *conntrack-state* = *State packet set*

The stateful primitive matcher: It is given the current state table. If match on *Established*, the packet must be known in the state table. If match on *New*, the packet must not be in the state table. If match on *IsSSH*, the dst port of the packet must be 22.

private fun *stateful-matcher* :: *conntrack-state* ⇒ (*primitive-matches*, *packet*) *matcher* **where**
stateful-matcher (*State state-table*) = (λ*m p*. *m* = *Established* ∧ *p* ∈ *state-table* ∨
m = *New* ∧ *p* ∉ *state-table* ∨
m = *IsSSH* ∧ (∃ *dst-port*. *p* = *SomePacket* (*22*, *dst-port*)))

Connections are always bi-directional.

private fun *reverse-direction* :: *packet* ⇒ *packet* **where**
reverse-direction OtherPacket = *OtherPacket* |
reverse-direction (*SomePacket* (*src*, *dst*)) = *SomePacket* (*dst*,*src*)

If a packet is accepted, the state for its bi-directional connection is saved in the state table.

private fun *state-update′* :: *conntrack-state* ⇒ *final-decision* ⇒ *packet* ⇒ *conntrack-state* **where**
state-update′ (*State state-table*) *FinalAllow p* = *State* (*state-table* ∪ {*p*, *reverse-direction p*}) |
state-update′ (*State state-table*) *FinalDeny p* = *State state-table*

Allow everything that is established and allow new ssh connections. Drop everything else (default policy, see below)

private definition *ruleset* == [*''INPUT''* ↦ [*Rule* (*Match Established*) *Accept*, *Rule* (*MatchAnd* (*Match IsSSH*) (*Match New*)) *Accept*]]

The *ruleset* does not allow *OtherPacket*

lemma *semantics-stateful ruleset stateful-matcher state-update′* (*State* {}) (*''INPUT''*, *Drop*) []
[(*OtherPacket*, *FinalDeny*)] (*State* {})
unfolding *ruleset-def*
apply(*rule semantics-stateful-intro-process-one*)
apply(*simp-all*)
apply(*rule semantics-stateful-intro-start*)
apply(*simp-all*)
apply(*rule seq-cons*)
apply(*rule call-result*)

apply(*simp-all*)
apply(*rule seq-cons*)
apply(*auto intro*: *iptables-bigstep.intros*)
done

The *ruleset* allows ssh packets, i.e. any packets with destination port 22 in the *New* rule. The state is updated such that everything which belongs to the connection will now be accepted.

lemma *semantics-stateful ruleset stateful-matcher state-update'* (*State* {}) (*''INPUT''*, *Drop*)
[]
[(*SomePacket* (*22*, *1024*), *FinalAllow*)]
(*State* {*SomePacket* (*1024*, *22*), *SomePacket* (*22*, *1024*)})
unfolding *ruleset-def*
apply(*rule semantics-stateful-intro-process-one*)
apply(*simp-all*)
apply(*rule semantics-stateful-intro-start*)
apply(*simp-all*)
apply(*rule seq-cons*)
apply(*rule call-result*)
apply(*simp-all*)
apply(*rule seq-cons*)
apply(*auto intro*: *iptables-bigstep.intros*)
done

If we continue with this state, answer packets are now allowed

lemma *semantics-stateful ruleset stateful-matcher state-update'* (*State* {}) (*''INPUT''*, *Drop*)
[]
[(*SomePacket* (*22*, *1024*), *FinalAllow*), (*SomePacket* (*1024*, *22*), *FinalAllow*)]
(*State* {*SomePacket* (*1024*, *22*), *SomePacket* (*22*, *1024*)})
unfolding *ruleset-def*
apply(*rule semantics-stateful-intro-process-one*)
apply(*simp-all*)
apply(*rule semantics-stateful-intro-process-one*)
apply(*simp-all*)
apply(*rule semantics-stateful-intro-start*)
apply(*simp-all*)
apply(*rule seq-cons*, *rule call-result*, *simp-all*, *rule seq-cons*)
apply(*auto intro*: *iptables-bigstep.intros*)
apply(*rule seq-cons*, *rule call-result*, *simp-all*, *rule seq-cons*)
apply(*auto intro*: *iptables-bigstep.intros*)
done

In contrast, without having previously established a state, answer packets are prohibited

If we continue with this state, answer packets are now allowed

lemma *semantics-stateful ruleset stateful-matcher state-update′* (*State* {}) (*''INPUT''*, *Drop*)
[]
[(*SomePacket* (*1024*, *22*), *FinalDeny*), (*SomePacket* (*22*, *1024*), *FinalAllow*), (*SomePacket* (*1024*, *22*), *FinalAllow*)]
(*State* {*SomePacket* (*1024*, *22*), *SomePacket* (*22*, *1024*)})
unfolding *ruleset-def*
apply(*rule semantics-stateful-intro-process-one*)
apply(*simp-all*)
apply(*rule semantics-stateful-intro-process-one*)
apply(*simp-all*)
apply(*rule semantics-stateful-intro-process-one*)
apply(*simp-all*)
apply(*rule semantics-stateful-intro-start*)
apply(*simp-all*)
apply(*rule seq-cons*, *rule call-result*, *simp-all*, *rule seq-cons*, *auto intro*: *iptables-bigstep.intros*)+
done

16.4 Example: Conntrack with packet tagging

datatype *packet-tag* = *TagNew* | *TagEstablished*
datatype *packet-tagged* = *SomePacket-tagged nat* × *nat* × *packet-tag* | *OtherPacket-tagged packet-tag*

fun *get-packet-tag* :: *packet-tagged* ⇒ *packet-tag* **where**
get-packet-tag (*SomePacket-tagged* (-,-, *tag*)) = *tag* |
get-packet-tag (*OtherPacket-tagged tag*) = *tag*

definition *stateful-matcher-tagged* :: (*primitive-matches*, *packet-tagged*) *matcher* **where**
stateful-matcher-tagged ≡ λ*m p*. *m* = *Established* ∧ (*get-packet-tag p* = *TagEstablished*) ∨
m = *New* ∧ (*get-packet-tag p* = *TagNew*) ∨
m = *IsSSH* ∧ (∃ *dst-port tag*. *p* = *SomePacket-tagged* (*22*, *dst-port*, *tag*))

fun *calculate-packet-tag* :: *conntrack-state* ⇒ *packet* ⇒ *packet-tag* **where**
calculate-packet-tag (*State state-table*) *p* = (*if p* ∈ *state-table then TagEstablished else TagNew*)

fun *packet-tagger* :: *conntrack-state* ⇒ *packet* ⇒ *packet-tagged* **where**
packet-tagger σ (*SomePacket* (*s*,*d*)) = (*SomePacket-tagged* (*s*,*d*, *calculate-packet-tag* σ (*SomePacket* (*s*,*d*)))) |
packet-tagger σ *OtherPacket* = (*OtherPacket-tagged* (*calculate-packet-tag* σ *OtherPacket*))

If a packet is accepted, the state for its bi-directional connection is saved in the state table.

fun *state-update-tagged* :: *conntrack-state* ⇒ *final-decision* ⇒ *packet* ⇒ *conntrack-state* **where**
state-update-tagged (*State state-table*) *FinalAllow p* = *State* (*state-table* ∪ {*p*, *reverse-direction p*}) |
state-update-tagged (*State state-table*) *FinalDeny p* = *State state-table*

Both semantics are equal

lemma *semantics-stateful rs stateful-matcher state-update′ σ*$_0$ *start ps ps-processed σ′* =
semantics-stateful-packet-tagging rs stateful-matcher-tagged packet-tagger state-update′ σ$_0$ *start ps ps-processed σ′*
apply(*rule semantics-stateful-vs-tagged*)
apply(*intro allI*, *rename-tac m σ p*)
apply(*case-tac σ*)
apply(*case-tac p*)
apply(*simp-all add*: *stateful-matcher-tagged-def*)
apply *force*
done
end

end
theory *Semantics-Goto*
imports *Main Firewall-Common Common/List-Misc HOL−Library.LaTeXsugar*
begin

17 Big Step Semantics with Goto

We extend the iptables semantics to support the goto action. A goto directly continues processing at the start of the called chain. It does not change the call stack. In contrast to calls, goto does not return. Consequently, everything behind a matching goto cannot be reached.

This theory is structured as follows. Fist, the goto semantics are introduced. Then, we show that those semantics are deterministic. Finally, we present two methods to remove gotos. The first unfolds goto. The second replaces gotos with calls. Finally, since the goto rules makes all proofs quite ugly, we never mention the goto semantics again. As we have shown, we can get rid of the gotos easily, thus, we stick to the nicer iptables semantics without goto.

context
begin

17.1 Semantics

private type-synonym *'a ruleset* = *string* ⇀ *'a rule list*

private type-synonym (*'a*, *'p*) *matcher* = *'a* ⇒ *'p* ⇒ *bool*

qualified fun *matches* :: ('*a*, '*p*) *matcher* ⇒ '*a match-expr* ⇒ '*p* ⇒ *bool* **where**
matches γ (*MatchAnd e1 e2*) *p* ⟷ *matches* γ *e1 p* ∧ *matches* γ *e2 p* |
matches γ (*MatchNot me*) *p* ⟷ ¬ *matches* γ *me p* |
matches γ (*Match e*) *p* ⟷ γ *e p* |
matches - *MatchAny* - ⟷ *True*

qualified fun *no-matching-Goto* :: ('*a*, '*p*) *matcher* ⇒ '*p* ⇒ '*a rule list* ⇒ *bool* **where**
no-matching-Goto - - [] ⟷ *True* |
no-matching-Goto γ *p* ((*Rule m* (*Goto* -))#*rs*) ⟷ ¬ *matches* γ *m p* ∧ *no-matching-Goto* γ *p rs* |
no-matching-Goto γ *p* (-#*rs*) ⟷ *no-matching-Goto* γ *p rs*

inductive *iptables-goto-bigstep* :: '*a ruleset* ⇒ ('*a*, '*p*) *matcher* ⇒ '*p* ⇒ '*a rule list* ⇒ *state* ⇒ *state* ⇒ *bool*
(‹-,-,-⊢$_g$ ⟨-, -⟩ ⇒ -› [*60*,*60*,*60*,*20*,*98*,*98*] *89*)
for Γ **and** γ **and** *p* **where**
skip: Γ,γ,*p*⊢$_g$ ⟨[], *t*⟩ ⇒ *t* |
accept: *matches* γ *m p* ⟹ Γ,γ,*p*⊢$_g$ ⟨[*Rule m Accept*], *Undecided*⟩ ⇒ *Decision FinalAllow* |
drop: *matches* γ *m p* ⟹ Γ,γ,*p*⊢$_g$ ⟨[*Rule m Drop*], *Undecided*⟩ ⇒ *Decision FinalDeny* |
reject: *matches* γ *m p* ⟹ Γ,γ,*p*⊢$_g$ ⟨[*Rule m Reject*], *Undecided*⟩ ⇒ *Decision FinalDeny* |
log: *matches* γ *m p* ⟹ Γ,γ,*p*⊢$_g$ ⟨[*Rule m Log*], *Undecided*⟩ ⇒ *Undecided* |
empty: *matches* γ *m p* ⟹ Γ,γ,*p*⊢$_g$ ⟨[*Rule m Empty*], *Undecided*⟩ ⇒ *Undecided* |
nomatch: ¬ *matches* γ *m p* ⟹ Γ,γ,*p*⊢$_g$ ⟨[*Rule m a*], *Undecided*⟩ ⇒ *Undecided* |
decision: Γ,γ,*p*⊢$_g$ ⟨*rs*, *Decision X*⟩ ⇒ *Decision X* |
seq: ⟦Γ,γ,*p*⊢$_g$ ⟨*rs*$_1$, *Undecided*⟩ ⇒ *t*; Γ,γ,*p*⊢$_g$ ⟨*rs*$_2$, *t*⟩ ⇒ *t*′; *no-matching-Goto* γ *p rs*$_1$⟧ ⟹ Γ,γ,*p*⊢$_g$ ⟨*rs*$_1$@*rs*$_2$, *Undecided*⟩ ⇒ *t*′ |
call-return: ⟦ *matches* γ *m p*; Γ *chain* = *Some* (*rs*$_1$@[*Rule m*′ *Return*]@*rs*$_2$);
matches γ *m*′ *p*; Γ,γ,*p*⊢$_g$ ⟨*rs*$_1$, *Undecided*⟩ ⇒ *Undecided*;
no-matching-Goto γ *p rs*$_1$⟧ ⟹
— we do not support a goto in the first part if you want to return
— probably unhandled case:
— `main:`
— `  call foo`
— `foo:`
— `  goto bar`
— `bar:`
— Return //returns to `call foo`
— But this would be a really awkward ruleset!
Γ,γ,*p*⊢$_g$ ⟨[*Rule m* (*Call chain*)], *Undecided*⟩ ⇒ *Undecided* |
call-result: ⟦ *matches* γ *m p*; Γ *chain* = *Some rs*; Γ,γ,*p*⊢$_g$ ⟨*rs*, *Undecided*⟩ ⇒ *t*

$\rrbracket \Longrightarrow$
$\Gamma,\gamma,p\vdash_g \langle[\mathit{Rule}\ m\ (\mathit{Call}\ \mathit{chain})],\ \mathit{Undecided}\rangle \Rightarrow t$ | — goto handling here seems okay
goto-decision: $[\![\ \mathit{matches}\ \gamma\ m\ p;\ \Gamma\ \mathit{chain} = \mathit{Some}\ \mathit{rs};\ \Gamma,\gamma,p\vdash_g \langle \mathit{rs},\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ X\ ]\!] \Longrightarrow$
$\Gamma,\gamma,p\vdash_g \langle(\mathit{Rule}\ m\ (\mathit{Goto}\ \mathit{chain}))\#\mathit{rest},\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ X$ |
goto-no-decision: $[\![\ \mathit{matches}\ \gamma\ m\ p;\ \Gamma\ \mathit{chain} = \mathit{Some}\ \mathit{rs};\ \Gamma,\gamma,p\vdash_g \langle \mathit{rs},\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}\ ]\!] \Longrightarrow$
$\Gamma,\gamma,p\vdash_g \langle(\mathit{Rule}\ m\ (\mathit{Goto}\ \mathit{chain}))\#\mathit{rest},\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}$

The semantic rules again in pretty format:

$$\frac{}{\Gamma,\gamma,p\vdash_g \langle[],\ t\rangle \Rightarrow t}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[\mathit{Rule}\ m\ \mathit{Accept}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalAllow}}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[\mathit{Rule}\ m\ \mathit{Drop}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalDeny}}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[\mathit{Rule}\ m\ \mathit{Reject}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalDeny}}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[\mathit{Rule}\ m\ \mathit{Log}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[\mathit{Rule}\ m\ \mathit{Empty}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}}$$

$$\frac{\neg\ \mathit{matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[\mathit{Rule}\ m\ a],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}}$$

$$\Gamma,\gamma,p\vdash_g \langle \mathit{rs},\ \mathit{Decision}\ X\rangle \Rightarrow \mathit{Decision}\ X$$

$$\frac{\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_1,\ \mathit{Undecided}\rangle \Rightarrow t \qquad \Gamma,\gamma,p\vdash_g \langle \mathit{rs}_2,\ t\rangle \Rightarrow t' \qquad \mathit{no\text{-}matching\text{-}Goto}\ \gamma\ p\ \mathit{rs}_1}{\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_1\ @\ \mathit{rs}_2,\ \mathit{Undecided}\rangle \Rightarrow t'}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p \qquad \Gamma\ \mathit{chain} = \mathit{Some}\ (\mathit{rs}_1\ @\ [\mathit{Rule}\ m'\ \mathit{Return}]\ @\ \mathit{rs}_2) \qquad \mathit{matches}\ \gamma\ m'\ p \qquad \Gamma,\gamma,p\vdash_g \langle \mathit{rs}_1,\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided} \qquad \mathit{no\text{-}matching\text{-}Goto}\ \gamma\ p\ \mathit{rs}_1}{\Gamma,\gamma,p\vdash_g \langle[\mathit{Rule}\ m\ (\mathit{Call}\ \mathit{chain})],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p \qquad \Gamma\ \mathit{chain} = \mathit{Some}\ \mathit{rs} \qquad \Gamma,\gamma,p\vdash_g \langle \mathit{rs},\ \mathit{Undecided}\rangle \Rightarrow t}{\Gamma,\gamma,p\vdash_g \langle[\mathit{Rule}\ m\ (\mathit{Call}\ \mathit{chain})],\ \mathit{Undecided}\rangle \Rightarrow t}$$

$$\frac{\mathit{matches}\ \gamma\ m\ p \qquad \Gamma\ \mathit{chain} = \mathit{Some}\ \mathit{rs} \qquad \Gamma,\gamma,p\vdash_g \langle \mathit{rs},\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ X}{\Gamma,\gamma,p\vdash_g \langle \mathit{Rule}\ m\ (\mathit{Goto}\ \mathit{chain})\cdot \mathit{rest},\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ X}$$

$$\frac{\Gamma\ chain = Some\ rs \qquad \begin{array}{c} matches\ \gamma\ m\ p \\ \Gamma,\gamma,p\vdash_g \langle rs,\ Undecided\rangle \Rightarrow Undecided \end{array}}{\Gamma,\gamma,p\vdash_g \langle Rule\ m\ (Goto\ chain)\cdot rest,\ Undecided\rangle \Rightarrow Undecided}$$

private lemma *deny*:
matches γ m $p \Longrightarrow a = Drop \vee a = Reject \Longrightarrow$ *iptables-goto-bigstep* Γ γ p [*Rule m a*] *Undecided* (*Decision FinalDeny*)
by (*auto intro*: *drop reject*)

private lemma *iptables-goto-bigstep-induct*
[*case-names*
Skip Allow Deny Log Nomatch Decision Seq Call-return Call-result Goto-Decision Goto-no-Decision,
induct pred: *iptables-goto-bigstep*]:
$[\![\ \Gamma,\gamma,p\vdash_g \langle rs,s\rangle \Rightarrow t;$
$\bigwedge t.\ P\ []\ t\ t;$
$\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Accept \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalAllow);$
$\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Drop \vee a = Reject \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalDeny);$
$\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Log \vee a = Empty \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided;$
$\bigwedge m\ a.\ \neg\ matches\ \gamma\ m\ p \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided;$
$\bigwedge rs\ X.\ P\ rs\ (Decision\ X)\ (Decision\ X);$
$\bigwedge rs\ rs_1\ rs_2\ t\ t'.\ rs = rs_1\ @\ rs_2 \Longrightarrow \Gamma,\gamma,p\vdash_g \langle rs_1,Undecided\rangle \Rightarrow t \Longrightarrow P\ rs_1\ Undecided\ t \Longrightarrow$
$\Gamma,\gamma,p\vdash_g \langle rs_2,t\rangle \Rightarrow t' \Longrightarrow P\ rs_2\ t\ t' \Longrightarrow no\text{-}matching\text{-}Goto\ \gamma\ p\ rs_1 \Longrightarrow$
$P\ rs\ Undecided\ t';$
$\bigwedge m\ a\ chain\ rs_1\ m'\ rs_2.\ matches\ \gamma\ m\ p \Longrightarrow a = Call\ chain \Longrightarrow$
$\Gamma\ chain = Some\ (rs_1\ @\ [Rule\ m'\ Return]\ @\ rs_2) \Longrightarrow$
$matches\ \gamma\ m'\ p \Longrightarrow \Gamma,\gamma,p\vdash_g \langle rs_1,Undecided\rangle \Rightarrow Undecided \Longrightarrow$
$no\text{-}matching\text{-}Goto\ \gamma\ p\ rs_1 \Longrightarrow P\ rs_1\ Undecided\ Undecided \Longrightarrow$
$P\ [Rule\ m\ a]\ Undecided\ Undecided;$
$\bigwedge m\ a\ chain\ rs\ t.\ matches\ \gamma\ m\ p \Longrightarrow a = Call\ chain \Longrightarrow \Gamma\ chain = Some\ rs \Longrightarrow$
$\Gamma,\gamma,p\vdash_g \langle rs,Undecided\rangle \Rightarrow t \Longrightarrow P\ rs\ Undecided\ t \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ t;$
$\bigwedge m\ a\ chain\ rs\ rest\ X.\ matches\ \gamma\ m\ p \Longrightarrow a = Goto\ chain \Longrightarrow \Gamma\ chain = Some\ rs \Longrightarrow$
$\Gamma,\gamma,p\vdash_g \langle rs,Undecided\rangle \Rightarrow (Decision\ X) \Longrightarrow P\ rs\ Undecided\ (Decision\ X) \Longrightarrow$
$P\ (Rule\ m\ a\#rest)\ Undecided\ (Decision\ X);$
$\bigwedge m\ a\ chain\ rs\ rest.\ matches\ \gamma\ m\ p \Longrightarrow a = Goto\ chain \Longrightarrow \Gamma\ chain = Some\ rs \Longrightarrow$
$\Gamma,\gamma,p\vdash_g \langle rs,Undecided\rangle \Rightarrow Undecided \Longrightarrow P\ rs\ Undecided$

Undecided $\Longrightarrow$

P (*Rule m a#rest*) *Undecided Undecided*$]\!] \Longrightarrow$

P rs s t

by (*induction rule*: *iptables-goto-bigstep.induct*) *auto*

17.1.1 Forward reasoning

private lemma *decisionD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow s = Decision\ X \Longrightarrow t = Decision\ X$

by (*induction rule*: *iptables-goto-bigstep-induct*) *auto*

private lemma *iptables-goto-bigstep-to-undecided*: $\Gamma,\gamma,p\vdash_g \langle rs,\ s\rangle \Rightarrow Undecided \Longrightarrow s = Undecided$

by (*metis decisionD state.exhaust*)

private lemma *iptables-goto-bigstep-to-decision*: $\Gamma,\gamma,p\vdash_g \langle rs,\ Decision\ Y\rangle \Rightarrow Decision\ X \Longrightarrow Y = X$

by (*metis decisionD state.inject*)

private lemma *skipD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [] \Longrightarrow s = t$

by (*induction rule*: *iptables-goto-bigstep.induct*) *auto*

private lemma *gotoD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ (Goto\ chain)] \Longrightarrow s = Undecided \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow$

$\exists\ rs.\ \Gamma\ chain = Some\ rs \wedge \Gamma,\gamma,p\vdash_g \langle rs,s\rangle \Rightarrow t$

by (*induction rule*: *iptables-goto-bigstep.induct*) (*auto dest*: *skipD elim*: *list-app-singletonE*)

private lemma *not-no-matching-Goto-singleton-cases*: $\neg$ *no-matching-Goto* γ *p* [*Rule m a*] $\longleftrightarrow (\exists\ chain.\ a = (Goto\ chain)) \wedge matches\ \gamma\ m\ p$

by(*case-tac a*) (*simp-all*)

private lemma *no-matching-Goto-Cons*: *no-matching-Goto* γ *p* [*r*] $\Longrightarrow$ *no-matching-Goto* γ *p rs* $\Longrightarrow$ *no-matching-Goto* γ *p* (*r#rs*)

by(*cases r*)(*rename-tac m a*, *case-tac a*, *simp-all*)

private lemma *no-matching-Goto-head*: *no-matching-Goto* γ *p* (*r#rs*) $\Longrightarrow$ *no-matching-Goto* γ *p* [*r*]

by(*cases r*)(*rename-tac m a*, *case-tac a*, *simp-all*)

private lemma *no-matching-Goto-tail*: *no-matching-Goto* γ *p* (*r#rs*) $\Longrightarrow$ *no-matching-Goto* γ *p rs*

by(*cases r*)(*rename-tac m a*, *case-tac a*, *simp-all*)

private lemma *not-no-matching-Goto-cases*:

assumes $\neg$ *no-matching-Goto* γ *p rs rs* $\neq$ []

shows $\exists$ *rs1 m chain rs2*. *rs* = *rs1*@(*Rule m* (*Goto chain*))#*rs2* $\wedge$ *no-matching-Goto* γ *p rs1* $\wedge$ *matches* γ *m p*

using *assms*

proof(*induction rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
note *Cons-outer=this*
from *Cons* **have** ¬ *no-matching-Goto* γ *p* (*r* # *rs*) **by** *simp*
show *?case*
proof(*cases rs*)
case *Nil*
obtain *m a* **where** *r* = *Rule m a* **by** (*cases r*) *simp*
with ‹¬ *no-matching-Goto* γ *p* (*r* # *rs*)› *Nil not-no-matching-Goto-singleton-cases*
have (∃ *chain*. *a* = (*Goto chain*)) ∧ *matches* γ *m p* **by** *metis*
from *this* **obtain** *chain* **where** *a* = (*Goto chain*) **and** *matches* γ *m p*
by *blast*
have *r* # *rs* = [] @ *Rule m* (*Goto chain*) # [] *no-matching-Goto* γ *p* []
matches γ *m p*
by (*simp-all add*: ‹*a* = *Goto chain*› ‹*r* = *Rule m a*› *Nil* ‹*matches* γ *m*
p›)
thus *?thesis* **by** *blast*
next
case(*Cons r′ rs′*)
with *Cons-outer* **have** *r* # *rs* = *r* # *r′* # *rs′* **by** *simp*
show *?thesis*
proof(*casesno-matching-Goto* γ *p* [*r*])
case *True*
with ‹¬ *no-matching-Goto* γ *p* (*r* # *rs*)› **have** ¬ *no-matching-Goto* γ
p rs **by** (*meson no-matching-Goto-Cons*)
have *rs* ≠ [] **using** *Cons* **by** *simp*
from *Cons-outer*(*1*)[*OF* ‹¬ *no-matching-Goto* γ *p rs*› ‹*rs* ≠ []›]
obtain *rs1 m chain rs2* **where** *rs* = *rs1* @ *Rule m* (*Goto chain*) #
rs2 no-matching-Goto γ *p rs1 matches* γ *m p* **by** *blast*
with ‹*r* # *rs* = *r* # *r′* # *rs′*› ‹*no-matching-Goto* γ *p* [*r*]›
no-matching-Goto-Cons
have *r* # *rs* = *r* # *rs1* @ *Rule m* (*Goto chain*) # *rs2* ∧
no-matching-Goto γ *p* (*r*#*rs1*) ∧ *matches* γ *m p* **by** *fast*
thus *?thesis*
apply(*rule-tac x*=*r*#*rs1* **in** *exI*)
by *auto*
next
case *False*
obtain *m a* **where** *r* = *Rule m a* **by** (*cases r*) *simp*
with *False not-no-matching-Goto-singleton-cases* **have** (∃ *chain*. *a* =
(*Goto chain*)) ∧ *matches* γ *m p* **by** *metis*
from *this* **obtain** *chain* **where** *a* = (*Goto chain*) **and** *matches* γ *m p*
by *blast*
have *r* # *rs* = [] @ *Rule m* (*Goto chain*) # *rs no-matching-Goto* γ *p*
[] *matches* γ *m p*
by (*simp-all add*: ‹*a* = *Goto chain*› ‹*r* = *Rule m a*› ‹*matches* γ *m p*›)
thus *?thesis* **by** *blast*

qed
qed
qed

private lemma *seq-cons-Goto-Undecided*:
assumes $\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ (Goto\ chain)],\ Undecided\rangle \Rightarrow Undecided$
and $\neg\ matches\ \gamma\ m\ p \Longrightarrow \Gamma,\gamma,p\vdash_g \langle rs,\ Undecided\rangle \Rightarrow Undecided$
shows $\Gamma,\gamma,p\vdash_g \langle Rule\ m\ (Goto\ chain)\ \#\ rs,\ Undecided\rangle \Rightarrow Undecided$
proof(*cases matches* $\gamma\ m\ p$)
case *True*
from *True assms* **have** $\exists rs.\ \Gamma\ chain = Some\ rs \wedge \Gamma,\gamma,p\vdash_g \langle rs,\ Undecided\rangle \Rightarrow Undecided$ **by**(*auto dest*: *gotoD*)
with *True* **show** *?thesis* **using** *goto-no-decision* **by** *fast*
next
case *False*
with *assms* **have** $\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ (Goto\ chain)]\ @\ rs,\ Undecided\rangle \Rightarrow Undecided$ **by**(*auto dest*: *seq*)
with *False* **show** *?thesis* **by** *simp*
qed

private lemma *seq-cons-Goto-t*:
$\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ (Goto\ chain)],\ Undecided\rangle \Rightarrow t \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow \Gamma,\gamma,p\vdash_g \langle Rule\ m\ (Goto\ chain)\ \#\ rs,\ Undecided\rangle \Rightarrow t$
apply(*frule gotoD*)
apply(*simp-all*)
apply(*clarify*)
apply(*cases t*)
apply(*auto intro*: *iptables-goto-bigstep.intros*)
done

private lemma *no-matching-Goto-append*: $no\text{-}matching\text{-}Goto\ \gamma\ p\ (rs1@rs2) \longleftrightarrow no\text{-}matching\text{-}Goto\ \gamma\ p\ rs1 \wedge no\text{-}matching\text{-}Goto\ \gamma\ p\ rs2$
by(*induction* $\gamma\ p\ rs1$ *rule*: *no-matching-Goto.induct*) (*simp-all*)

private lemma *no-matching-Goto-append1*: $no\text{-}matching\text{-}Goto\ \gamma\ p\ (rs1@rs2) \Longrightarrow no\text{-}matching\text{-}Goto\ \gamma\ p\ rs1$
using *no-matching-Goto-append* **by** *fast*
private lemma *no-matching-Goto-append2*: $no\text{-}matching\text{-}Goto\ \gamma\ p\ (rs1@rs2) \Longrightarrow no\text{-}matching\text{-}Goto\ \gamma\ p\ rs2$
using *no-matching-Goto-append* **by** *fast*

private lemma *seq-cons*:
assumes $\Gamma,\gamma,p\vdash_g \langle[r],Undecided\rangle \Rightarrow t$ **and** $\Gamma,\gamma,p\vdash_g \langle rs,t\rangle \Rightarrow t'$ **and** $no\text{-}matching\text{-}Goto\ \gamma\ p\ [r]$
shows $\Gamma,\gamma,p\vdash_g \langle r\#rs,\ Undecided\rangle \Rightarrow t'$
proof −
from *assms* **have** $\Gamma,\gamma,p\vdash_g \langle[r]\ @\ rs,\ Undecided\rangle \Rightarrow t'$ **by** (*rule seq*)
thus *?thesis* **by** *simp*

qed

context
notes *skipD*[*dest*] *list-app-singletonE*[*elim*]
begin
lemma *acceptD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Accept] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Decision\ FinalAllow$
by (*induction rule*: *iptables-goto-bigstep.induct*) *auto*

lemma *dropD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Drop] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Decision\ FinalDeny$
by (*induction rule*: *iptables-goto-bigstep.induct*) *auto*

lemma *rejectD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Reject] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Decision\ FinalDeny$
by (*induction rule*: *iptables-goto-bigstep.induct*) *auto*

lemma *logD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Log] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Undecided$
by (*induction rule*: *iptables-goto-bigstep.induct*) *auto*

lemma *emptyD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Empty] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Undecided$
by (*induction rule*: *iptables-goto-bigstep.induct*) *auto*

lemma *nomatchD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ a] \Longrightarrow s = Undecided \Longrightarrow \neg\ matches\ \gamma\ m\ p \Longrightarrow t = Undecided$
by (*induction rule*: *iptables-goto-bigstep.induct*) *auto*

lemma *callD*:
assumes $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t\ r = [Rule\ m\ (Call\ chain)]\ s = Undecided\ matches\ \gamma\ m\ p\ \Gamma\ chain = Some\ rs$
obtains $\Gamma,\gamma,p\vdash_g \langle rs,s\rangle \Rightarrow t$
$\mid rs_1\ rs_2\ m'$ **where** $rs = rs_1\ @\ Rule\ m'\ Return\ \#\ rs_2\ matches\ \gamma\ m'\ p\ \Gamma,\gamma,p\vdash_g \langle rs_1,s\rangle \Rightarrow Undecided\ no\text{-}matching\text{-}Goto\ \gamma\ p\ rs_1\ t = Undecided$
using *assms*
proof (*induction r s t arbitrary*: *rs rule*: *iptables-goto-bigstep.induct*)
case (*seq* rs_1)
thus *?case* **by** (*cases* rs_1) *auto*
qed *auto*
end

private lemmas *iptables-goto-bigstepD* = *skipD acceptD dropD rejectD logD emptyD nomatchD decisionD callD gotoD*

private lemma *seq'*:
assumes $rs = rs_1\ @\ rs_2\ \Gamma,\gamma,p\vdash_g \langle rs_1,s\rangle \Rightarrow t\ \Gamma,\gamma,p\vdash_g \langle rs_2,t\rangle \Rightarrow t'$ **and** *no-matching-Goto* $\gamma\ p\ rs_1$
shows $\Gamma,\gamma,p\vdash_g \langle rs,s\rangle \Rightarrow t'$

using *assms* **by** (*cases s*) (*auto intro*: *seq decision dest*: *decisionD*)

private lemma *seq'-cons*: $\Gamma,\gamma,p\vdash_g \langle[r],s\rangle \Rightarrow t \Longrightarrow \Gamma,\gamma,p\vdash_g \langle rs,t\rangle \Rightarrow t' \Longrightarrow$ *no-matching-Goto* γ p $[r] \Longrightarrow \Gamma,\gamma,p\vdash_g \langle r\#rs,\ s\rangle \Rightarrow t'$
by (*metis decision decisionD state.exhaust seq-cons*)

private lemma *no-matching-Goto-take*: *no-matching-Goto* γ p $rs \Longrightarrow$ *no-matching-Goto* γ p (*take n rs*)
apply(*induction n arbitrary*: *rs*)
apply(*simp-all*)
apply(*rename-tac r rs*)
apply(*case-tac rs*)
apply(*simp-all*)
apply(*rename-tac r' rs'*)
apply(*case-tac r'*)
apply(*simp*)
apply(*rename-tac m a*)
by(*case-tac a*) (*simp-all*)

private lemma *seq-split*:
assumes $\Gamma,\gamma,p\vdash_g \langle rs,\ s\rangle \Rightarrow t$ $rs = rs_1@rs_2$
obtains (*no-matching-Goto*) t' **where** $\Gamma,\gamma,p\vdash_g \langle rs_1,s\rangle \Rightarrow t'$ $\Gamma,\gamma,p\vdash_g \langle rs_2,t'\rangle \Rightarrow t$ *no-matching-Goto* γ p rs_1
| (*matching-Goto*) $\Gamma,\gamma,p\vdash_g \langle rs_1,s\rangle \Rightarrow t \neg$ *no-matching-Goto* γ p rs_1
proof −
have $(\exists t'.\ \Gamma,\gamma,p\vdash_g \langle rs_1,s\rangle \Rightarrow t' \wedge \Gamma,\gamma,p\vdash_g \langle rs_2,t'\rangle \Rightarrow t \wedge$ *no-matching-Goto* γ p $rs_1) \vee (\Gamma,\gamma,p\vdash_g \langle rs_1,s\rangle \Rightarrow t \wedge \neg$ *no-matching-Goto* γ p $rs_1)$
using *assms*
proof (*induction rs s t arbitrary*: rs_1 rs_2 *rule*: *iptables-goto-bigstep-induct*)
case *Skip* **thus** *?case* **by** (*auto intro*: *iptables-goto-bigstep.intros simp add*: *accept*)
next
case *Allow* **thus** *?case* **by** (*cases* rs_1) (*auto intro*: *iptables-goto-bigstep.intros simp add*: *accept*)
next
case *Deny* **thus** *?case* **by** (*cases* rs_1) (*auto intro*: *iptables-goto-bigstep.intros simp add*: *deny*)
next
case *Log* **thus** *?case* **by** (*cases* rs_1) (*auto intro*: *iptables-goto-bigstep.intros simp add*: *log empty*)
next
case *Nomatch* **thus** *?case* **by** (*cases* rs_1)
(*auto intro*: *iptables-goto-bigstep.intros simp add*: *not-no-matching-Goto-singleton-cases*, *meson nomatch not-no-matching-Goto-singleton-cases skip*)
next
case *Decision* **thus** *?case* **by** (*auto intro*: *iptables-goto-bigstep.intros*)
next

case (*Seq rs rsa rsb t t'*)
hence *rs*: *rsa* @ *rsb* = rs_1 @ rs_2 **by** *simp*
note *List.append-eq-append-conv-if*[*simp*]
from *rs* **show** *?case*
proof (*cases rule*: *list-app-eq-cases*)
case *longer*
with *Seq* **have** *t1*: Γ,γ,*p*⊢$_g$ ⟨*take* (*length rsa*) rs_1, *Undecided*⟩ ⇒ *t*
by *simp*
from *Seq.IH*(*2*)[*OF longer*(*2*)] **have** *IH*:
(∃ *t'a*. Γ,γ,*p*⊢$_g$ ⟨*drop* (*length rsa*) rs_1, *t*⟩ ⇒ *t'a* ∧ Γ,γ,*p*⊢$_g$ ⟨rs_2, *t'a*⟩ ⇒ *t'* ∧ *no-matching-Goto* γ *p* (*drop* (*length rsa*) rs_1)) ∨
Γ,γ,*p*⊢$_g$ ⟨*drop* (*length rsa*) rs_1, *t*⟩ ⇒ *t'* ∧ ¬ *no-matching-Goto* γ *p* (*drop* (*length rsa*) rs_1) (**is** *?IH-no-Goto* ∨ *?IH-Goto*) **by** *simp*
thus *?thesis*
proof(*rule disjE*)
assume *IH*: *?IH-no-Goto*
from *IH* **obtain** *t2*
where *t2a*: Γ,γ,*p*⊢$_g$ ⟨*drop* (*length rsa*) rs_1,*t*⟩ ⇒ *t2*
and *rs-part2*: Γ,γ,*p*⊢$_g$ ⟨rs_2,*t2*⟩ ⇒ *t'*
and *no-matching-Goto* γ *p* (*drop* (*length rsa*) rs_1)
by *blast*
with *t1 rs-part2* **have** *rs-part1*: Γ,γ,*p*⊢$_g$ ⟨*take* (*length rsa*) rs_1 @ *drop* (*length rsa*) rs_1, *Undecided*⟩ ⇒ *t2*
using *Seq.hyps*(*4*) *longer*(*1*) *seq* **by** *fastforce*
have *no-matching-Goto* γ *p* (*take* (*length rsa*) rs_1 @ *drop* (*length rsa*) rs_1)
using *Seq.hyps*(*4*) ‹*no-matching-Goto* γ *p* (*drop* (*length rsa*) rs_1)› *longer*(*1*)
no-matching-Goto-append **by** *fastforce*
with *Seq rs-part1 rs-part2* **show** *?thesis* **by** *auto*
next
assume *?IH-Goto*
thus *?thesis* **by** (*metis Seq.hyps*(*2*) *Seq.hyps*(*4*) *append-take-drop-id longer*(*1*) *no-matching-Goto-append2 seq'*)
qed
next
case *shorter*
from *shorter rs* **have** *rsa'*: *rsa* = rs_1 @ *take* (*length rsa* − *length* rs_1) rs_2
by (*metis append-eq-conv-conj length-drop*)
from *shorter rs* **have** *rsb'*: *rsb* = *drop* (*length rsa* − *length* rs_1) rs_2
by (*metis append-eq-conv-conj length-drop*)

from *Seq.hyps*(*4*) *rsa' no-matching-Goto-append2* **have**
no-matching-Goto-rs2: *no-matching-Goto* γ *p* (*take* (*length rsa* − *length* rs_1) rs_2) **by** *metis*

from *rsb' Seq.hyps* **have** *t2*: Γ,γ,*p*⊢$_g$ ⟨*drop* (*length rsa* − *length* rs_1) rs_2,*t*⟩ ⇒ *t'*
by *blast*

```
          from Seq.IH(1)[OF rsa'] have IH:
              (∃ t'. Γ,γ,p⊢_g ⟨rs_1, Undecided⟩ ⇒ t' ∧ Γ,γ,p⊢_g ⟨take (length rsa −
length rs_1) rs_2, t'⟩ ⇒ t ∧ no-matching-Goto γ p rs_1) ∨
                Γ,γ,p⊢_g ⟨rs_1, Undecided⟩ ⇒ t ∧ ¬ no-matching-Goto γ p rs_1 (is
?IH-no-Goto ∨ ?IH-Goto) by simp

          thus ?thesis
            proof(rule disjE)
              assume IH: ?IH-no-Goto
              from IH obtain t1
                where t1a: Γ,γ,p⊢_g ⟨rs_1,Undecided⟩ ⇒ t1
                  and t1b: Γ,γ,p⊢_g ⟨take (length rsa − length rs_1) rs_2,t1⟩ ⇒ t
                  and no-matching-Goto γ p rs_1
                by blast

                from no-matching-Goto-rs2 t2 seq' t1b have rs2: Γ,γ,p⊢_g ⟨rs_2,t1⟩
⇒ t'
                  by fastforce
                from t1a rs2 ‹no-matching-Goto γ p rs_1› show ?thesis by fast
              next
                assume ?IH-Goto
                thus ?thesis by (metis Seq.hyps(4) no-matching-Goto-append1 rsa')
              qed
          qed
        next
          case Call-return
          hence Γ,γ,p⊢_g ⟨rs_1, Undecided⟩ ⇒ Undecided Γ,γ,p⊢_g ⟨rs_2, Undecided⟩ ⇒
Undecided
         by (case-tac [!] rs_1) (auto intro: iptables-goto-bigstep.skip iptables-goto-bigstep.call-return)
          thus ?case by fast
        next
          case (Call-result - - - - t)
          show ?case
            proof (cases rs_1)
              case Nil
               with Call-result have Γ,γ,p⊢_g ⟨rs_1, Undecided⟩ ⇒ Undecided Γ,γ,p⊢_g
⟨rs_2, Undecided⟩ ⇒ t
                by (auto intro: iptables-goto-bigstep.intros)
              thus ?thesis using local.Nil by auto
            next
              case Cons
             with Call-result have Γ,γ,p⊢_g ⟨rs_1, Undecided⟩ ⇒ t Γ,γ,p⊢_g ⟨rs_2, t⟩ ⇒ t
                by (auto intro: iptables-goto-bigstep.intros)
              thus ?thesis by fast
            qed
        next
          case (Goto-Decision m a chain rs rest X)
          thus ?case
```

```
        proof (cases rs1)
          case Nil
        with Goto-Decision have Γ,γ,p⊢g ⟨rs1, Undecided⟩ ⇒ Undecided Γ,γ,p⊢g
⟨rs2, Undecided⟩ ⇒ Decision X
            by (auto intro: iptables-goto-bigstep.intros)
          thus ?thesis using local.Nil by auto
        next
          case Cons
        with Goto-Decision have Γ,γ,p⊢g ⟨rs1, Undecided⟩ ⇒ Decision X Γ,γ,p⊢g
⟨rs2, Decision X⟩ ⇒ Decision X
            by (auto intro: iptables-goto-bigstep.intros)
          thus ?thesis by fast
        qed
      next
        case (Goto-no-Decision m a chain rs rest rs1)
        from Goto-no-Decision have rs1rs2: Rule m (Goto chain) # rest = rs1 @
rs2 by simp
          from goto-no-decision[OF Goto-no-Decision(1)]  Goto-no-Decision(3)
Goto-no-Decision(4)
          have x: ⋀rest. Γ,γ,p⊢g ⟨Rule m (Goto chain) # rest, Undecided⟩ ⇒
Undecided by simp
        show ?case
          proof (cases rs1)
            case Nil
              with Goto-no-Decision have Γ,γ,p⊢g ⟨rs1, Undecided⟩ ⇒ Undecided
Γ,γ,p⊢g ⟨rs2, Undecided⟩ ⇒ Undecided
              by (auto intro: iptables-goto-bigstep.intros)
            thus ?thesis by fast
          next
            case (Cons rs1a rs1s)
          with rs1rs2 have rs1 = Rule m (Goto chain) # (take (length rs1s) rest)
by simp
            from Cons rs1rs2 haversrs2 = drop (length rs1s) rest by simp

              from Cons Goto-no-Decision have 1: Γ,γ,p⊢g ⟨rs1, Undecided⟩ ⇒
Undecided
              using x by auto[1]
            have 2: ¬ no-matching-Goto γ p rs1
              by (simp add: Goto-no-Decision.hyps(1) ‹rs1 = Rule m (Goto chain)
# take (length rs1s) rest›)
            from 1 2 show ?thesis by fast
          qed
      qed
    thus ?thesis using matching-Goto no-matching-Goto by blast
    qed

    private lemma seqE:
      assumes Γ,γ,p⊢g ⟨rs1@rs2, s⟩ ⇒ t
      obtains (no-matching-Goto) ti where Γ,γ,p⊢g ⟨rs1,s⟩ ⇒ ti Γ,γ,p⊢g ⟨rs2,ti⟩
```

$\Rightarrow$ *t no-matching-Goto* γ *p* rs_1
| (*matching-Goto*) Γ,γ,*p*$\vdash_g$ $\langle rs_1,s\rangle \Rightarrow t \neg$ *no-matching-Goto* γ *p* rs_1
using *assms* **by** (*force elim*: *seq-split*)

private lemma *seqE-cons*:
assumes Γ,γ,*p*$\vdash_g$ $\langle r\#rs,\ s\rangle \Rightarrow t$
obtains (*no-matching-Goto*) *ti* **where** Γ,γ,*p*$\vdash_g$ $\langle[r],s\rangle \Rightarrow$ *ti* Γ,γ,*p*$\vdash_g$ $\langle rs,ti\rangle \Rightarrow$ *t no-matching-Goto* γ *p* $[r]$
| (*matching-Goto*) Γ,γ,*p*$\vdash_g$ $\langle[r],s\rangle \Rightarrow t \neg$ *no-matching-Goto* γ *p* $[r]$
using *assms* **by** (*metis append-Cons append-Nil seqE*)

private lemma *seqE-cons-Undecided*:
assumes Γ,γ,*p*$\vdash_g$ $\langle r\#rs,\ Undecided\rangle \Rightarrow t$
obtains (*no-matching-Goto*) *ti* **where** Γ,γ,*p*$\vdash_g$ $\langle[r],Undecided\rangle \Rightarrow$ *ti* **and** Γ,γ,*p*$\vdash_g$ $\langle rs,ti\rangle \Rightarrow t$ **and** *no-matching-Goto* γ *p* $[r]$
| (*matching-Goto*) *m chain rs′* **where** $r =$ *Rule m* (*Goto chain*) **and** Γ,γ,*p*$\vdash_g$ $\langle[$*Rule m* (*Goto chain*)$],Undecided\rangle \Rightarrow t$ **and** *matches* γ *m p* Γ *chain* = *Some rs′*
using *assms*
proof(*cases rule*: *seqE-cons*)
case *no-matching-Goto* **thus** *?thesis* **using** *local.that* **by** *simp*
next
case *matching-Goto*
from *this*(*2*) *not-no-matching-Goto-singleton-cases*[*of* γ *p* (*get-match r*) (*get-action r*), *simplified*] **have**
$((\exists$ *chain*. (*get-action r*) = *Goto chain*$) \wedge$ *matches* γ (*get-match r*) *p*) **by** *simp*
from *this* **obtain** *chain m* **where** *r*: $r =$ *Rule m* (*Goto chain*) *matches* γ *m p* **by**(*cases r*) *auto*
from *matching-Goto r* **have** Γ,γ,*p*$\vdash_g$ $\langle[$*Rule m* (*Goto chain*)$],Undecided\rangle \Rightarrow t$ **by** *simp*
from *gotoD*[*OF matching-Goto*(*1*)] *r* ‹*matches* γ *m p*› **obtain** *rs′* **where** Γ *chain* = *Some rs′* **by** *blast*
from *local.that*
show *?thesis* **using** ‹Γ *chain* = *Some rs′*› ‹Γ,γ,*p*$\vdash_g$ $\langle[$*Rule m* (*Goto chain*)$]$, *Undecided*$\rangle \Rightarrow t$› *r*(*1*) *r*(*2*) **by** *blast*
qed

private lemma *nomatch′*:
assumes $\bigwedge r.\ r \in$ *set rs* $\Longrightarrow \neg$ *matches* γ (*get-match r*) *p*
shows Γ,γ,*p*$\vdash_g$ $\langle rs,\ s\rangle \Rightarrow s$
proof(*cases s*)
case *Undecided*
have $\forall\, r{\in}$*set rs*. $\neg$ *matches* γ (*get-match r*) *p* $\Longrightarrow \Gamma$,γ,*p*$\vdash_g$ $\langle rs,\ Undecided\rangle \Rightarrow$ *Undecided*
proof(*induction rs*)
case *Nil*
thus *?case* **by** (*fast intro*: *skip*)
next

case (*Cons r rs*)
hence Γ,γ,p⊢$_g$ ⟨[*r*], *Undecided*⟩ ⇒ *Undecided*
by (*cases r*) (*auto intro*: *nomatch*)
with *Cons* **show** *?case*
by (*metis list.set-intros*(*1*) *list.set-intros*(*2*) *not-no-matching-Goto-singleton-cases rule.collapse seq'-cons*)
qed
with *assms Undecided* **show** *?thesis* **by** *simp*
qed (*blast intro*: *decision*)

private lemma *no-free-return*: **assumes** Γ,γ,p⊢$_g$ ⟨[*Rule m Return*], *Undecided*⟩ ⇒ *t* **and** *matches* γ *m p* **shows** *False*
proof −
{ fix *a s*
have *no-free-return-hlp*: Γ,γ,p⊢$_g$ ⟨*a*,*s*⟩ ⇒ *t* ⟹ *matches* γ *m p* ⟹ *s* = *Undecided* ⟹ *a* = [*Rule m Return*] ⟹ *False*
proof (*induction rule*: *iptables-goto-bigstep.induct*)
case (*seq rs*$_1$)
thus *?case*
by (*cases rs*$_1$) (*auto dest*: *skipD*)
qed *simp-all*
} with *assms* **show** *?thesis* **by** *blast*
qed

17.2 Determinism

private lemma *iptables-goto-bigstep-Undecided-Undecided-deterministic*:
Γ,γ,p⊢$_g$ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided* ⟹ Γ,γ,p⊢$_g$ ⟨*rs*, *Undecided*⟩ ⇒ *t* ⟹ *t* = *Undecided*
proof(*induction rs Undecided Undecided arbitrary*: *t rule*: *iptables-goto-bigstep-induct*)
case *Skip* **thus** *?case* **by**(*fastforce dest*: *skipD logD emptyD nomatchD decisionD*)
next
case *Log* **thus** *?case* **by**(*fastforce dest*: *skipD logD emptyD nomatchD decisionD*)
next
case *Nomatch* **thus** *?case* **by**(*fastforce dest*: *skipD logD emptyD nomatchD decisionD*)
next
case *Seq* **thus** *?case* **by** (*metis iptables-goto-bigstep-to-undecided seqE*)
next
case (*Call-return m a chain rs*$_1$ *m'* *rs*$_2$)
from *Call-return* **have** Γ,γ,p⊢$_g$ ⟨[*Rule m* (*Call chain*)], *Undecided*⟩ ⇒ *Undecided*
apply(*frule-tac rs*$_1$=*rs*$_1$ **and** *m'*=*m'* **and** *chain*=*chain* **in** *call-return*)
by(*simp-all*)
with *Call-return* **show** *?case*
apply *simp*
apply (*metis callD no-free-return seqE seqE-cons*)

done
next
case *Call-result* **thus** *?case* **by** (*meson callD*)
next
case *Goto-no-Decision* **thus** *?case* **by** (*metis gotoD no-matching-Goto.simps*(*2*) *option.sel seqE-cons*)
qed

private lemma *iptables-goto-bigstep-Undecided-deterministic*:
$\Gamma,\gamma,p\vdash_g \langle rs,\ Undecided\rangle \Rightarrow t \Longrightarrow \Gamma,\gamma,p\vdash_g \langle rs,\ Undecided\rangle \Rightarrow t' \Longrightarrow\ t' = t$
proof(*induction rs Undecided t arbitrary*: *t' rule*: *iptables-goto-bigstep-induct*)
case *Skip* **thus** *?case* **by**(*fastforce dest*: *skipD logD emptyD nomatchD decisionD*)
next
case *Allow* **thus** *?case* **by** (*auto dest*: *iptables-goto-bigstepD*)
next
case *Deny* **thus** *?case* **by** (*auto dest*: *iptables-goto-bigstepD*)
next
case *Log* **thus** *?case* **by** (*auto dest*: *iptables-goto-bigstepD*)
next
case *Nomatch* **thus** *?case* **by** (*auto dest*: *iptables-goto-bigstepD*)
next
case *Seq* **thus** *?case* **by** (*metis decisionD seqE state.exhaust*)
next
case *Call-return* **thus** *?case* **by** (*meson call-return iptables-goto-bigstep-Undecided-Undecided-deterministic*
next
case *Call-result* **thus** *?case* **by** (*metis callD call-result iptables-goto-bigstep-Undecided-Undecided-determini*
next
case *Goto-Decision* **thus** *?case* **by** (*metis gotoD no-matching-Goto.simps*(*2*) *option.sel seqE-cons*)
next
case *Goto-no-Decision* **thus** *?case* **by** (*meson goto-no-decision iptables-goto-bigstep-Undecided-Undecided-d*
qed

qualified theorem *iptables-goto-bigstep-deterministic*: **assumes** $\Gamma,\gamma,p\vdash_g \langle rs,\ s\rangle \Rightarrow t$ **and** $\Gamma,\gamma,p\vdash_g \langle rs,\ s\rangle \Rightarrow t'$ **shows** $t = t'$
using *assms*
apply(*cases s*)
apply(*simp add*: *iptables-goto-bigstep-Undecided-deterministic*)
by(*auto dest*: *decisionD*)

17.3 Matching

private lemma *matches-rule-and-simp-help*:
assumes *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash_g \langle[Rule\ (MatchAnd\ m\ m')\ a'],\ s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle[Rule\ m'\ a'],\ s\rangle \Rightarrow t$ (**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *?l* **thus** *?r*

by (*induction* [*Rule* (*MatchAnd m m'*) *a'*] *s t rule*: *iptables-goto-bigstep-induct*)
(*auto intro*: *iptables-goto-bigstep.intros simp*: *assms Cons-eq-append-conv dest*: *skipD*)
next
assume *?r* **thus** *?l*
by (*induction* [*Rule m' a'*] *s t rule*: *iptables-goto-bigstep-induct*)
(*auto intro*: *iptables-goto-bigstep.intros simp*: *assms Cons-eq-append-conv dest*: *skipD*)
qed

private lemma *matches-MatchNot-simp*:
assumes *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash_g \langle[Rule\ (MatchNot\ m)\ a], Undecided\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle[], Undecided\rangle \Rightarrow t$ (**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *?l* **thus** *?r*
by (*induction* [*Rule* (*MatchNot m*) *a*] *Undecided t rule*: *iptables-goto-bigstep-induct*)
(*auto intro*: *iptables-goto-bigstep.intros simp*: *assms Cons-eq-append-conv dest*: *skipD*)
next
assume *?r*
hence *t* = *Undecided*
by (*metis skipD*)
with *assms* **show** *?l*
by (*fastforce intro*: *nomatch*)
qed

private lemma *matches-MatchNotAnd-simp*:
assumes *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash_g \langle[Rule\ (MatchAnd\ (MatchNot\ m)\ m')\ a], Undecided\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle[], Undecided\rangle \Rightarrow t$ (**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *?l* **thus** *?r*
by (*induction* [*Rule* (*MatchAnd* (*MatchNot m*) *m'*) *a*] *Undecided t rule*: *iptables-goto-bigstep-induct*)
(*auto intro*: *iptables-goto-bigstep.intros simp add*: *assms Cons-eq-append-conv dest*: *skipD*)
next
assume *?r*
hence *t* = *Undecided*
by (*metis skipD*)
with *assms* **show** *?l*
by (*fastforce intro*: *nomatch*)
qed

private lemma *matches-rule-and-simp*:
assumes *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash_g \langle[Rule\ (MatchAnd\ m\ m')\ a'], s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle[Rule\ m'\ a'], s\rangle \Rightarrow t$

proof (*cases s*)
case *Undecided*
with *assms* **show** *?thesis*
by (*simp add*: *matches-rule-and-simp-help*)
next
case *Decision*
thus *?thesis* **by** (*metis decision decisionD*)
qed

qualified definition *add-match* :: *'a match-expr* ⇒ *'a rule list* ⇒ *'a rule list* **where**
add-match m rs = *map* (λ*r*. *case r of Rule m' a'* ⇒ *Rule* (*MatchAnd m m'*) *a'*) *rs*

private lemma *add-match-split*: *add-match m* (*rs1*@*rs2*) = *add-match m rs1* @ *add-match m rs2*
unfolding *add-match-def*
by (*fact map-append*)

private lemma *add-match-split-fst*: *add-match m* (*Rule m' a'* # *rs*) = *Rule* (*MatchAnd m m'*) *a'* # *add-match m rs*
unfolding *add-match-def*
by *simp*

private lemma *matches-add-match-no-matching-Goto-simp*: *matches* γ *m p* ⟹ *no-matching-Goto* γ *p* (*add-match m rs*) ⟹ *no-matching-Goto* γ *p rs*
apply(*induction rs*)
apply(*simp-all*)
apply(*rename-tac r rs*)
apply(*case-tac r*)
apply(*simp add*: *add-match-split-fst no-matching-Goto-tail*)
apply(*drule no-matching-Goto-head*)
apply(*rename-tac m' a'*)
apply(*case-tac a'*)
apply *simp-all*
done

private lemma *matches-add-match-no-matching-Goto-simp2*: *matches* γ *m p* ⟹ *no-matching-Goto* γ *p rs* ⟹ *no-matching-Goto* γ *p* (*add-match m rs*)
apply(*induction rs*)
apply(*simp add*: *add-match-def*)
apply(*rename-tac r rs*)
apply(*case-tac r*)
apply(*simp add*: *add-match-split-fst no-matching-Goto-tail*)
apply(*rename-tac m' a'*)
apply(*case-tac a'*)

```
        apply simp-all
    done

  private lemma matches-add-match-MatchNot-no-matching-Goto-simp: ¬ matches
γ m p ⟹ no-matching-Goto γ p (add-match m rs)
    apply(induction rs)
     apply(simp add: add-match-def)
    apply(rename-tac r rs)
    apply(case-tac r)
    apply(simp add: add-match-split-fst no-matching-Goto-tail)
    apply(rename-tac m' a')
    apply(case-tac a')
        apply simp-all
    done


   private lemma not-matches-add-match-simp:
    assumes ¬ matches γ m p
    shows Γ,γ,p⊢g ⟨add-match m rs, Undecided⟩ ⇒ t ⟷ Γ,γ,p⊢g ⟨[], Undecided⟩
⇒ t
    proof(induction rs)
    case Nil thus ?case unfolding add-match-def by simp
    next
    case (Cons r rs)
     obtain m' a where r: r = Rule m' a by(cases r, simp)
     let ?lhs=Γ,γ,p⊢g ⟨Rule (MatchAnd m m') a # add-match m rs, Undecided⟩
⇒ t
      let ?rhs=Γ,γ,p⊢g ⟨[], Undecided⟩ ⇒ t
      { assume ?lhs
        from ‹?lhs› Cons have ?rhs
          proof(cases  Γ γ p Rule (MatchAnd m m') a add-match m rs  t rule:
seqE-cons-Undecided)
         case (no-matching-Goto ti)
          hence ti = Undecided  by (simp add: assms nomatchD)
          with no-matching-Goto Cons show ?thesis by simp
         next
         case (matching-Goto) with Cons assms show ?thesis by force
        qed
      } note 1=this
      { assume ?rhs
        hence t = Undecided using skipD by metis
        with Cons.IH ‹?rhs› have ?lhs
       by (meson assms matches.simps(1) nomatch not-no-matching-Goto-singleton-cases
seq-cons)
      } with 1 show ?case by(auto simp add: r add-match-split-fst)
    qed

   private lemma matches-add-match-MatchNot-simp:
    assumes m: matches γ m p
```

shows $\Gamma,\gamma,p\vdash_g \langle \mathit{add\text{-}match}\ (\mathit{MatchNot}\ m)\ rs,\ s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle [],\ s\rangle \Rightarrow t$
(**is** *?l s* $\longleftrightarrow$ *?r s*)
proof (*cases s*)
case *Undecided*
have *?l Undecided* $\longleftrightarrow$ *?r Undecided*
proof
assume *?l Undecided* **with** *m* **show** *?r Undecided*
proof (*induction rs*)
case *Nil*
thus *?case*
unfolding *add-match-def* **by** *simp*
next
case (*Cons r rs*)
thus *?case*
by (*cases r*) (*metis matches-MatchNotAnd-simp skipD seqE-cons add-match-split-fst*)
qed
next
assume *?r Undecided* **with** *m* **show** *?l Undecided*
proof (*induction rs*)
case *Nil*
thus *?case*
unfolding *add-match-def* **by** *simp*
next
case (*Cons r rs*)
hence *t* = *Undecided* **using** *skipD* **by** *metis*
with *Cons* **show** *?case*
apply (*cases r*)
apply(*simp add*: *add-match-split-fst*)
by (*metis matches.simps*(*1*) *matches.simps*(*2*) *matches-MatchNotAnd-simp not-no-matching-Goto-singleton-cases seq-cons*)
qed
qed
with *Undecided* **show** *?thesis* **by** *fast*
next
case (*Decision d*)
thus *?thesis*
by(*metis decision decisionD*)
qed

private lemma *just-show-all-bigstep-semantics-equalities-with-start-Undecided*:

$\Gamma,\gamma,p\vdash_g \langle rs1,\ \mathit{Undecided}\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle rs2,\ \mathit{Undecided}\rangle \Rightarrow t \Longrightarrow$
$\Gamma,\gamma,p\vdash_g \langle rs1,\ s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle rs2,\ s\rangle \Rightarrow t$
apply(*cases s*)
apply(*simp*)
apply(*simp*)
using *decision decisionD* **by** *fastforce*

```
  private lemma matches-add-match-simp-helper:
    assumes m: matches γ m p
   shows Γ,γ,p⊢_g ⟨add-match m rs, Undecided⟩ ⇒ t ⟷ Γ,γ,p⊢_g ⟨rs, Undecided⟩
⇒ t (is ?l ⟷ ?r)
    proof
      assume ?l with m show ?r
        proof (induction rs)
          case Nil
          thus ?case
            unfolding add-match-def by simp
        next
          case (Cons r rs)
           obtain m' a where r: r = Rule m' a by(cases r, simp)
            from Cons have  Γ,γ,p⊢_g ⟨Rule (MatchAnd m m') a # add-match m
rs, Undecided⟩ ⇒ t
              by(simp add: r add-match-split-fst)
           from this Cons have Γ,γ,p⊢_g ⟨Rule m' a # rs, Undecided⟩ ⇒ t
           proof(cases rule: seqE-cons-Undecided)
             case (no-matching-Goto ti)
          from no-matching-Goto(3) Cons.prems(1) not-no-matching-Goto-singleton-cases
              have no-matching-Goto γ p [Rule m' a] by (metis matches.simps(1))
              with no-matching-Goto Cons show ?thesis
               apply(simp add: matches-rule-and-simp)
               apply(cases ti)
                apply (simp add: seq'-cons)
               by (metis decision decisionD seq'-cons)
             next
             case (matching-Goto) with Cons show ?thesis
              apply(clarify)
              apply(simp add: matches-rule-and-simp-help)
              by (simp add: seq-cons-Goto-t)
           qed
           thus ?case by(simp add: r)
        qed
    next
      assume ?r with m show ?l
        proof (induction rs)
          case Nil
          thus ?case
            unfolding add-match-def by simp
        next
          case (Cons r rs)
           obtain m' a where r: r = Rule m' a by(cases r, simp)
            from Cons have Γ,γ,p⊢_g ⟨Rule m' a # rs, Undecided⟩ ⇒ t by(simp
add: r)
           from this have Γ,γ,p⊢_g ⟨Rule (MatchAnd m m') a # add-match m rs,
Undecided⟩ ⇒ t
             proof(cases Γ γ p Rule m' a rs t rule: seqE-cons-Undecided)
```

```
            case (no-matching-Goto ti)
          from no-matching-Goto Cons.prems matches-rule-and-simp[symmetric]
have
              Γ,γ,p⊢g ⟨[Rule (MatchAnd m m') a], Undecided⟩ ⇒ ti by fast
            with Cons.prems Cons.IH no-matching-Goto show ?thesis
             apply(cases ti)
           apply (metis matches.simps(1) not-no-matching-Goto-singleton-cases
seq-cons)
          apply (metis decision decisionD matches.simps(1) not-no-matching-Goto-singleton-cases
seq-cons)
             done
           next
           case (matching-Goto) with Cons show ?thesis
            by (simp add: matches-rule-and-simp-help seq-cons-Goto-t)
         qed
         thus ?case by(simp add: r add-match-split-fst)
       qed
     qed


   private lemma matches-add-match-simp:
    matches γ m p ⟹ Γ,γ,p⊢g ⟨add-match m rs, s⟩ ⇒ t ⟷ Γ,γ,p⊢g ⟨rs, s⟩ ⇒ t
     apply(rule just-show-all-bigstep-semantics-equalities-with-start-Undecided)
     by(simp add: matches-add-match-simp-helper)

   private lemma not-matches-add-matchNot-simp:
      ¬ matches γ m p ⟹ Γ,γ,p⊢g ⟨add-match (MatchNot m) rs, s⟩ ⇒ t ⟷
Γ,γ,p⊢g ⟨rs, s⟩ ⇒ t
     by (simp add: matches-add-match-simp)
```

17.4 Goto Unfolding

```
   private lemma unfold-Goto-Undecided:
        assumes chain-defined: Γ chain = Some rs and no-matching-Goto-rs:
no-matching-Goto γ p rs
      shows Γ,γ,p⊢g ⟨(Rule m (Goto chain))#rest, Undecided⟩ ⇒ t ⟷ Γ,γ,p⊢g
⟨add-match m rs @ add-match (MatchNot m) rest, Undecided⟩ ⇒ t
            (is ?l ⟷ ?r)
   proof
     assume ?l
     thus ?r
       proof(cases rule: seqE-cons-Undecided)
       case (no-matching-Goto ti)
         from no-matching-Goto have ¬ matches γ m p by simp
           with no-matching-Goto have ti: ti = Undecided using nomatchD by
metis
         from ‹¬ matches γ m p› have Γ,γ,p⊢g ⟨add-match m rs, Undecided⟩ ⇒
Undecided
          using not-matches-add-match-simp skip by fast
```

from ‹¬ *matches* γ *m p*› *matches-add-match-MatchNot-no-matching-Goto-simp* **have** *no-matching-Goto* γ *p* (*add-match m rs*) **by** *force*
from *no-matching-Goto ti* **have** $\Gamma,\gamma,p\vdash_g \langle rest, Undecided\rangle \Rightarrow t$ **by** *simp*
with *not-matches-add-matchNot-simp*[*OF* ‹¬ *matches* γ *m p*›] **have** $\Gamma,\gamma,p\vdash_g$ ⟨*add-match* (*MatchNot m*) *rest*, *Undecided*⟩ ⇒ *t* **by** *simp*
show *?thesis*
by (*meson* ‹$\Gamma,\gamma,p\vdash_g$ ⟨*add-match* (*MatchNot m*) *rest*, *Undecided*⟩ ⇒ *t*› ‹$\Gamma,\gamma,p\vdash_g$ ⟨*add-match m rs*, *Undecided*⟩ ⇒ *Undecided*› ‹*no-matching-Goto* γ *p* (*add-match m rs*)› *seq*)
next
case (*matching-Goto m chain rs'*)
from *matching-Goto gotoD assms* **have** $\Gamma,\gamma,p\vdash_g \langle rs, Undecided\rangle \Rightarrow t$ **by** *fastforce*
hence *1*: $\Gamma,\gamma,p\vdash_g$ ⟨*add-match m rs*, *Undecided*⟩ ⇒ *t* **by** (*simp add*: *matches-add-match-simp matching-Goto*(*3*))
have *2*: $\Gamma,\gamma,p\vdash_g$ ⟨*add-match* (*MatchNot m*) *rest*, *t*⟩ ⇒ *t* **by** (*simp add*: *matches-add-match-MatchNot-simp matching-Goto*(*3*) *skip*)
from *no-matching-Goto-rs matches-add-match-no-matching-Goto-simp2 matching-Goto* **have** *3*: *no-matching-Goto* γ *p* (*add-match m rs*) **by** *fast*
from *1 2 3* **show** *?thesis* **using** *matching-Goto*(*1*) *seq* **by** *fastforce*
qed
next
assume *?r*
thus *?l*
proof(*cases matches* γ *m p*)
case *True*
have $\Gamma,\gamma,p\vdash_g \langle rs, Undecided\rangle \Rightarrow t$
by (*metis True* ‹$\Gamma,\gamma,p\vdash_g$ ⟨*add-match m rs* @ *add-match* (*MatchNot m*) *rest*, *Undecided*⟩ ⇒ *t*›
matches-add-match-MatchNot-simp matches-add-match-simp-helper self-append-conv seq' seqE)
show *?l*
apply(*cases t*)
using *goto-no-decision*[*OF True*] *chain-defined* **apply** (*metis* ‹$\Gamma,\gamma,p\vdash_g$ ⟨*rs*, *Undecided*⟩ ⇒ *t*›)
using *goto-decision*[*OF True*, *of* Γ *chain rs - rest*] *chain-defined* **apply** (*metis* ‹$\Gamma,\gamma,p\vdash_g$ ⟨*rs*, *Undecided*⟩ ⇒ *t*›)
done
next
case *False*
with ‹*?r*› **have** $\Gamma,\gamma,p\vdash_g$ ⟨*add-match* (*MatchNot m*) *rest*, *Undecided*⟩ ⇒ *t*
by (*metis matches-add-match-MatchNot-no-matching-Goto-simp not-matches-add-match-simp seqE skipD*)
with *False* **have** $\Gamma,\gamma,p\vdash_g \langle rest, Undecided\rangle \Rightarrow t$ **by** (*meson not-matches-add-matchNot-simp*)
show *?l* **by** (*meson False* ‹$\Gamma,\gamma,p\vdash_g$ ⟨*rest*, *Undecided*⟩ ⇒ *t*› *nomatch not-no-matching-Goto-singleton-cases seq-cons*)
qed
qed

qualified theorem *unfold-Goto*:
assumes *chain-defined*: Γ *chain* = *Some rs* **and** *no-matching-Goto-rs*: *no-matching-Goto* γ *p rs*
shows Γ,γ,*p*$\vdash_g$ $\langle$(*Rule m* (*Goto chain*))#*rest*, *s*$\rangle$ $\Rightarrow$ *t* $\longleftrightarrow$ Γ,γ,*p*$\vdash_g$ $\langle$*add-match m rs* @ *add-match* (*MatchNot m*) *rest*, *s*$\rangle$ $\Rightarrow$ *t*
apply(*rule just-show-all-bigstep-semantics-equalities-with-start-Undecided*)
using *assms unfold-Goto-Undecided* **by** *fast*

A chain that will definitely come to a direct decision

qualified fun *terminal-chain* :: *'a rule list* $\Rightarrow$ *bool* **where**
terminal-chain [] = *False* |
terminal-chain [*Rule MatchAny Accept*] = *True* |
terminal-chain [*Rule MatchAny Drop*] = *True* |
terminal-chain [*Rule MatchAny Reject*] = *True* |
terminal-chain ((*Rule* - (*Goto* -))#*rs*) = *False* |
terminal-chain ((*Rule* - (*Call* -))#*rs*) = *False* |
terminal-chain ((*Rule* - *Return*)#*rs*) = *False* |
terminal-chain ((*Rule* - *Unknown*)#*rs*) = *False* |
terminal-chain (-#*rs*) = *terminal-chain rs*

private lemma *terminal-chain-no-matching-Goto*: *terminal-chain rs* $\Longrightarrow$ *no-matching-Goto* γ *p rs*
by(*induction rs rule*: *terminal-chain.induct*) *simp-all*

A terminal chain means (if the semantics are actually defined) that the chain will ultimately yield a final filtering decision, for all packets.

qualified lemma *terminal-chain rs* $\Longrightarrow$ Γ,γ,*p*$\vdash_g$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *t* $\Longrightarrow$ $\exists$ *X*. *t* = *Decision X*
apply(*induction rs*)
apply(*simp*)
apply(*rename-tac r rs*)
apply(*case-tac r*)
apply(*rename-tac m a*)
apply(*simp*)
apply(*frule-tac* γ=γ **and** *p*=*p* **in** *terminal-chain-no-matching-Goto*)
apply(*case-tac a*)
apply(*simp-all*)
apply(*erule seqE-cons*, *simp-all*,
metis iptables-goto-bigstepD matches.elims terminal-chain.simps terminal-chain.simps terminal-chain.simps)+
done

private lemma *replace-Goto-with-Call-in-terminal-chain-Undecided*:
assumes *chain-defined*: Γ *chain* = *Some rs* **and** *terminal-chain*: *terminal-chain rs*

shows $\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ (Goto\ chain)],\ Undecided\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow t$
(**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *?l*
thus *?r*
proof(*cases rule*: *seqE-cons-Undecided*)
case (*no-matching-Goto ti*)
from *no-matching-Goto* **have** $\neg$ *matches* γ *m p* **by** *simp*
with *nomatch* **have** *1*: $\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ (Goto\ chain)],\ Undecided\rangle \Rightarrow Undecided$ **by** *fast*
from ‹$\neg$ *matches* γ *m p*› *nomatch* **have** *2*: $\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow Undecided$ **by** *fast*
from *1 2* **show** *?thesis*
using ‹*?l*› *iptables-goto-bigstep-Undecided-Undecided-deterministic* **by** *fastforce*
next
case (*matching-Goto m chain rs'*)
from *matching-Goto gotoD assms* **have** $\Gamma,\gamma,p\vdash_g \langle rs,\ Undecided\rangle \Rightarrow t$ **by** *fastforce*
from *call-result*[*OF* ‹*matches* γ *m p*› *chain-defined* ‹$\Gamma,\gamma,p\vdash_g \langle rs,\ Undecided\rangle \Rightarrow t$›] **show** *?thesis*
by (*metis matching-Goto*(*1*) *rule.sel*(*1*))
qed
next
assume *?r*
thus *?l*
proof(*cases matches* γ *m p*)
case *True*
{**fix** *rs1*::*'a rule list* **and** *m'* **and** *rs2*
have *terminal-chain* (*rs1* @ *Rule m' Return* # *rs2*) $\Longrightarrow$ *False*
apply(*induction rs1*)
apply(*simp-all*)
apply(*rename-tac r' rs'*)
apply(*case-tac r'*)
apply(*rename-tac m a*)
apply(*simp-all*)
apply(*case-tac a*)
apply(*simp-all*)
apply (*metis append-is-Nil-conv hd-Cons-tl terminal-chain.simps*)+
done
} **note** *no-return=this*
have $\Gamma,\gamma,p\vdash_g \langle rs,\ Undecided\rangle \Rightarrow t$
apply(*rule callD*[*OF* ‹*?r*› - - *True chain-defined*])
apply(*simp-all*)
using *no-return terminal-chain* **by** *blast*
show *?l*
apply(*cases t*)
using *goto-no-decision*[*OF True*] *chain-defined* **apply** (*metis* ‹$\Gamma,\gamma,p\vdash_g$

```
⟨rs, Undecided⟩ ⇒ t›)
            using goto-decision[OF True, of Γ chain rs - []] chain-defined apply
(metis ‹Γ,γ,p⊢g ⟨rs, Undecided⟩ ⇒ t›)
          done
        next
        case False
          show ?l using False ‹Γ,γ,p⊢g ⟨[Rule m (Call chain)], Undecided⟩ ⇒ t›
nomatch nomatchD by fastforce
        qed
      qed


  qualified theorem replace-Goto-with-Call-in-terminal-chain:
        assumes chain-defined: Γ chain = Some rs and terminal-chain: termi-
nal-chain rs
       shows Γ,γ,p⊢g ⟨[Rule m (Goto chain)], s⟩ ⇒ t ⟷ Γ,γ,p⊢g ⟨[Rule m (Call
chain)], s⟩ ⇒ t
     apply(rule just-show-all-bigstep-semantics-equalities-with-start-Undecided)
     using assms replace-Goto-with-Call-in-terminal-chain-Undecided by fast


  qualified fun rewrite-Goto-chain-safe :: (string ⇀ 'a rule list) ⇒ 'a rule list ⇒
('a rule list) option where
    rewrite-Goto-chain-safe - [] = Some [] |
    rewrite-Goto-chain-safe Γ ((Rule m (Goto chain))#rs) =
      (case (Γ chain) of None     ⇒ None
                  | Some rs' ⇒ (if
                                  ¬ terminal-chain rs'
                               then
                                  None
                               else
                                     map-option (λrs. Rule m (Call chain) # rs)
(rewrite-Goto-chain-safe Γ rs)
                              )
      ) |
   rewrite-Goto-chain-safe Γ (r#rs) = map-option (λrs. r # rs) (rewrite-Goto-chain-safe
Γ rs)

  private fun rewrite-Goto-safe-internal
    :: (string × 'a rule list) list ⇒ (string × 'a rule list) list ⇒ (string × 'a rule
list) list option where
    rewrite-Goto-safe-internal - [] = Some [] |
    rewrite-Goto-safe-internal Γ ((chain-name, rs)#cs) =
             (case rewrite-Goto-chain-safe (map-of Γ) rs of
                    None ⇒ None
                       | Some rs' ⇒ map-option (λrst. (chain-name, rs')#rst)
(rewrite-Goto-safe-internal Γ cs)
             )
```

qualified fun *rewrite-Goto-safe* :: (*string* × *'a rule list*) *list* ⇒ (*string* × *'a rule list*) *list option* **where**
rewrite-Goto-safe cs = *rewrite-Goto-safe-internal cs cs*

qualified definition *rewrite-Goto* :: (*string* × *'a rule list*) *list* ⇒ (*string* × *'a rule list*) *list* **where**
rewrite-Goto cs = *the* (*rewrite-Goto-safe cs*)

private lemma *step-IH-cong*: (⋀*s*. Γ,γ,*p*⊢$_g$ ⟨*rs1*, *s*⟩ ⇒ *t* = Γ,γ,*p*⊢$_g$ ⟨*rs2*, *s*⟩ ⇒ *t*) ⟹
Γ,γ,*p*⊢$_g$ ⟨*r*#*rs1*, *s*⟩ ⇒ *t* = Γ,γ,*p*⊢$_g$ ⟨*r*#*rs2*, *s*⟩ ⇒ *t*
apply(*rule iffI*)
apply(*erule seqE-cons*)
apply(*rule seq'-cons*)
apply *simp-all*
apply(*drule not-no-matching-Goto-cases*)
apply(*simp*; *fail*)
apply(*elim exE conjE*, *rename-tac rs1a m chain rs2a*)
apply(*subgoal-tac r* = *Rule m* (*Goto chain*))
prefer *2*
subgoal by (*simp add*: *Cons-eq-append-conv*)
apply(*thin-tac* [*r*] = - @ *Rule m* (*Goto chain*) # -)
apply *simp*
apply (*metis decision decisionD seq-cons-Goto-t state.exhaust*)
apply(*erule seqE-cons*)
apply(*rule seq'-cons*)
apply *simp-all*
apply(*drule not-no-matching-Goto-cases*)
apply(*simp*; *fail*)
apply(*elim exE conjE*, *rename-tac rs1a m chain rs2a*)
apply(*subgoal-tac r* = *Rule m* (*Goto chain*))
prefer *2*
subgoal by (*simp add*: *Cons-eq-append-conv*)
apply(*thin-tac* [*r*] = - @ *Rule m* (*Goto chain*) # -)
apply *simp*
apply (*metis decision decisionD seq-cons-Goto-t state.exhaust*)
done

private lemma *terminal-chain-decision*:
terminal-chain rs ⟹ Γ,γ,*p*⊢$_g$ ⟨*rs*, *Undecided*⟩ ⇒ *t* ⟹ ∃ *X*. *t* = *Decision X*
apply(*induction rs arbitrary*: *t rule*: *terminal-chain.induct*)
apply *simp-all*
apply(*auto dest*: *iptables-goto-bigstepD*)[*3*]
apply(*erule seqE-cons*, *simp-all*, *blast dest*: *iptables-goto-bigstepD*)+
done

private lemma *terminal-chain-Goto-decision*: Γ *chain* = *Some rs* $\Longrightarrow$ *terminal-chain rs* $\Longrightarrow$ *matches* γ *m p* $\Longrightarrow$
Γ,γ,*p*$\vdash_g$ $\langle$[*Rule m* (*Goto chain*)], *s*$\rangle$ $\Rightarrow$ *t* $\Longrightarrow$ $\exists$ *X*. *t* = *Decision X*
apply(*cases s*)
apply(*drule gotoD*, *simp-all*)
apply(*elim exE conjE*, *simp-all*)
using *terminal-chain-decision* **apply** *fast*
by (*meson decisionD*)

qualified theorem *rewrite-Goto-chain-safe*:
rewrite-Goto-chain-safe Γ *rs* = *Some rs'* $\Longrightarrow$ Γ,γ,*p*$\vdash_g$ $\langle$*rs'*, *s*$\rangle$ $\Rightarrow$ *t* $\longleftrightarrow$ Γ,γ,*p*$\vdash_g$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow$ *t*
proof(*induction* Γ *rs arbitrary*: *rs' s rule*: *rewrite-Goto-chain-safe.induct*)
case *1* **thus** *?case* **by** (*simp split*: *option.split-asm if-split-asm*)
next
case (*2* Γ *m chain rs*)
from *2*(*2*) **obtain** *z x2* **where** Γ *chain* = *Some x2* **and** *terminal-chain x2*
and *rs'* = *Rule m* (*Call chain*) # *z*
and *Some z* = *rewrite-Goto-chain-safe* Γ *rs*
by(*auto split*: *option.split-asm if-split-asm*)
from *2*(*1*) ‹Γ *chain* = *Some x2*› ‹*terminal-chain x2*› ‹*Some z* = *rewrite-Goto-chain-safe* Γ *rs*›
have *IH*: Γ,γ,*p*$\vdash_g$ $\langle$*z*, *s*$\rangle$ $\Rightarrow$ *t* = Γ,γ,*p*$\vdash_g$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow$ *t* **for** *s* **by** *simp*

have Γ,γ,*p*$\vdash_g$ $\langle$*Rule m* (*Call chain*) # *z*, *Undecided*$\rangle$ $\Rightarrow$ *t* $\longleftrightarrow$ Γ,γ,*p*$\vdash_g$ $\langle$*Rule m* (*Goto chain*) # *rs*, *Undecided*$\rangle$ $\Rightarrow$ *t*
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*intro iffI*)
assume *?lhs*
with *IH* **obtain** *ti* **where** *ti1*: Γ,γ,*p*$\vdash_g$ $\langle$[*Rule m* (*Call chain*)], *Undecided*$\rangle$ $\Rightarrow$ *ti* **and** *ti2*: Γ,γ,*p*$\vdash_g$ $\langle$*rs*, *ti*$\rangle$ $\Rightarrow$ *t*
by(*auto elim*: *seqE-cons*)
show *?rhs*
proof(*cases matches* γ *m p*)
case *False*
from *replace-Goto-with-Call-in-terminal-chain* ‹Γ *chain* = *Some x2*› ‹*terminal-chain x2*›
have Γ,γ,*p*$\vdash_g$ $\langle$[*Rule m* (*Call chain*)], *Undecided*$\rangle$ $\Rightarrow$ *ti* $\longleftrightarrow$ Γ,γ,*p*$\vdash_g$ $\langle$[*Rule m* (*Goto chain*)], *Undecided*$\rangle$ $\Rightarrow$ *ti*
by *fast*
with *False ti1 ti2* **show** *?thesis* **by**(*rule-tac t=ti* **in** *seq'-cons*) *simp+*
next
case *True*
from *ti1* ‹Γ *chain* = *Some x2*› ‹*terminal-chain x2*›
have *g*: Γ,γ,*p*$\vdash_g$ $\langle$[*Rule m* (*Goto chain*)], *Undecided*$\rangle$ $\Rightarrow$ *ti*
by(*subst*(*asm*) *replace-Goto-with-Call-in-terminal-chain*[*symmetric*]) *simp+*

```
      with True ‹Γ chain = Some x2› ‹terminal-chain x2› obtain X where X:
ti = Decision X
        by(blast dest: terminal-chain-Goto-decision)
      with this ti2 have t = Decision X
        by(simp add: decisionD)
      with g X True ti2 ‹Γ chain = Some x2› ‹terminal-chain x2› show ?thesis
        apply(simp)
        apply(rule seq-cons-Goto-t, simp-all)
        done
    qed
  next
    assume ?rhs
   with IH ‹Γ chain = Some x2› ‹terminal-chain x2› ‹Some z = rewrite-Goto-chain-safe
Γ rs› show ?lhs
      apply −
      apply(erule seqE-cons)
       subgoal for ti
       apply simp-all
       apply(rule-tac t=ti in seq'-cons)
        apply simp-all
       using replace-Goto-with-Call-in-terminal-chain by fast
      apply simp
      apply(frule(3) terminal-chain-Goto-decision)
      apply(subst(asm) replace-Goto-with-Call-in-terminal-chain, simp-all)
      apply(rule seq'-cons, simp-all)
      apply(elim exE)
      by (simp add: decision)
  qed
  with ‹rs' = Rule m (Call chain) # z› show ?case
    apply −
    apply(rule just-show-all-bigstep-semantics-equalities-with-start-Undecided)
    by simp

 qed(auto cong: step-IH-cong)
```

Example: The semantics are actually defined (for this example).

```
 lemma defines γ ≡ (λ- -. True) and m ≡ MatchAny
 shows [''FORWARD'' ↦ [Rule m Log, Rule m (Call ''foo''), Rule m Drop],
        ''foo'' ↦ [Rule m Log, Rule m (Goto ''bar''), Rule m Reject],
        ''bar'' ↦ [Rule m (Goto ''baz''), Rule m Reject],
        ''baz'' ↦ [(Rule m Accept)]],
     γ,p⊢_g⟨[Rule MatchAny (Call ''FORWARD'')], Undecided⟩ ⇒ (Decision Fi-
nalAllow)
 apply(subgoal-tac matches γ m p)
  prefer 2
  apply(simp add: γ-def m-def; fail)
 apply(rule call-result)
   apply(auto)
 apply(rule-tac t=Undecided in seq-cons)
```

apply(*auto intro*: *log*)
apply(*rule-tac t*=*Decision FinalAllow* **in** *seq-cons*)
apply(*auto intro*: *decision*)
apply(*rule call-result*)
apply(*simp*)+
apply(*rule-tac t*=*Undecided* **in** *seq-cons*)
apply(*auto intro*: *log*)
apply(*rule goto-decision*)
apply(*simp*)+
apply(*rule goto-decision*)
apply(*simp*)+
apply(*auto intro*: *accept*)
done

end

end

18 Negation Type DNF

theory *Negation-Type-DNF*
imports *Negation-Type*
begin

type-synonym *'a dnf* = (('*a negation-type*) *list*) *list*

fun *cnf-to-bool* :: (*'a* ⇒ *bool*) ⇒ *'a negation-type list* ⇒ *bool* **where**
cnf-to-bool - [] ⟷ *True* |
cnf-to-bool f (*Pos a*#*as*) ⟷ (*f a*) ∧ *cnf-to-bool f as* |
cnf-to-bool f (*Neg a*#*as*) ⟷ (¬ *f a*) ∧ *cnf-to-bool f as*

fun *dnf-to-bool* :: (*'a* ⇒ *bool*) ⇒ *'a dnf* ⇒ *bool* **where**
dnf-to-bool - [] ⟷ *False* |
dnf-to-bool f (*as*#*ass*) ⟷ (*cnf-to-bool f as*) ∨ (*dnf-to-bool f ass*)

representing *True*

definition *dnf-True* :: *'a dnf* **where**
dnf-True ≡ [[]]
lemma *dnf-True*: *dnf-to-bool f dnf-True*
unfolding *dnf-True-def* **by**(*simp*)

representing *False*

definition *dnf-False* :: *'a dnf* **where**

dnf-False ≡ []
lemma *dnf-False*: ¬ *dnf-to-bool f dnf-False*
unfolding *dnf-False-def* **by**(*simp*)

lemma *cnf-to-bool-append*: *cnf-to-bool γ* (*a1* @ *a2*) ⟷ *cnf-to-bool γ a1* ∧ *cnf-to-bool γ a2*
by(*induction γ a1 rule*: *cnf-to-bool.induct*) (*simp-all*)
lemma *dnf-to-bool-append*: *dnf-to-bool γ* (*a1* @ *a2*) ⟷ *dnf-to-bool γ a1* ∨ *dnf-to-bool γ a2*
by(*induction a1*) (*simp-all*)

definition *dnf-and* :: *'a dnf* ⇒ *'a dnf* ⇒ *'a dnf* **where**
dnf-and cnf1 cnf2 = [*andlist1* @ *andlist2*. *andlist1* <− *cnf1*, *andlist2* <− *cnf2*]

value *dnf-and* ([[*a*,*b*], [*c*,*d*]]) ([[*v*,*w*], [*x*,*y*]])

lemma *cnf-to-bool-set*: *cnf-to-bool f cnf* ⟷ (∀ *c* ∈ *set cnf*. (*case c of Pos a* ⇒ *f a* | *Neg a* ⇒ ¬ *f a*))
proof(*induction cnf*)
case *Nil* **thus** *?case* **by** *simp*
next
case *Cons* **thus** *?case* **by** (*simp split*: *negation-type.split*)
qed
lemma *dnf-to-bool-set*: *dnf-to-bool γ dnf* ⟷ (∃ *d* ∈ *set dnf*. *cnf-to-bool γ d*)
proof(*induction dnf*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons d d1*) **thus** *?case* **by**(*simp*)
qed

lemma *dnf-to-bool-seteq*: *set ‘ set d1* = *set ‘ set d2* ⟹ *dnf-to-bool γ d1* ⟷ *dnf-to-bool γ d2*
proof −
assume *assm*: *set ‘ set d1* = *set ‘ set d2*
have *helper1*: ⋀*P d*. (∃ *d*∈*set d*. ∀ *c*∈*set d*. *P c*) ⟷ (∃ *d*∈*set ‘ set d*. ∀ *c*∈*d*. *P c*) **by** *blast*
from *assm* **show** *?thesis*
apply(*simp add*: *dnf-to-bool-set cnf-to-bool-set*)
apply(*subst helper1*)
apply(*subst helper1*)
apply(*simp*)
done
qed

lemma *dnf-and-correct*: *dnf-to-bool γ* (*dnf-and d1 d2*) ⟷ *dnf-to-bool γ d1* ∧ *dnf-to-bool γ d2*
apply(*simp add*: *dnf-and-def*)
apply(*induction d1*)
apply(*simp*)

apply(*simp add*: *dnf-to-bool-append*)
apply(*simp add*: *dnf-to-bool-set cnf-to-bool-set*)
by (*meson UnCI UnE*)

lemma *dnf-and-symmetric*: *dnf-to-bool* γ (*dnf-and d1 d2*) $\longleftrightarrow$ *dnf-to-bool* γ (*dnf-and d2 d1*)
using *dnf-and-correct* **by** *blast*

18.0.1 inverting a DNF

Example

lemma $(\neg\ ((a1 \wedge a2) \vee b \vee c)) = ((\neg a1 \wedge \neg\ b \wedge \neg\ c) \vee (\neg a2 \wedge \neg\ b \wedge \neg\ c))$ **by** *blast*
lemma $(\neg\ ((a1 \wedge a2) \vee (b1 \wedge b2) \vee c)) = ((\neg a1 \wedge \neg\ b1 \wedge \neg\ c) \vee (\neg a2 \wedge \neg\ b1 \wedge \neg\ c) \vee (\neg a1 \wedge \neg\ b2 \wedge \neg\ c) \vee (\neg a2 \wedge \neg\ b2 \wedge \neg\ c))$ **by** *blast*

fun *listprepend* :: *'a list* $\Rightarrow$ *'a list list* $\Rightarrow$ *'a list list* **where**
listprepend [] *ns* = [] |
listprepend (*a*#*as*) *ns* = (*map* (λ*xs*. *a*#*xs*) *ns*) @ (*listprepend as ns*)

lemma *listprepend* [*a*,*b*] [*as*, *bs*] = [*a*#*as*, *a*#*bs*, *b*#*as*, *b*#*bs*] **by** *simp*

lemma *map-a-and*: *dnf-to-bool* γ (*map* ((#) *a*) *ds*) $\longleftrightarrow$ *dnf-to-bool* γ [[*a*]] $\wedge$ *dnf-to-bool* γ *ds*
apply(*induction ds*)
apply(*simp-all*)
apply(*case-tac a*)
apply(*simp-all*)
apply *blast+*
done

this is how *listprepend* works:

lemma $\neg$ *dnf-to-bool* γ (*listprepend* [] *ds*) **by**(*simp*)
lemma *dnf-to-bool* γ (*listprepend* [*a*] *ds*) $\longleftrightarrow$ *dnf-to-bool* γ [[*a*]] $\wedge$ *dnf-to-bool* γ *ds* **by**(*simp add*: *map-a-and*)
lemma *dnf-to-bool* γ (*listprepend* [*a*, *b*] *ds*) $\longleftrightarrow$ (*dnf-to-bool* γ [[*a*]] $\wedge$ *dnf-to-bool* γ *ds*) $\vee$ (*dnf-to-bool* γ [[*b*]] $\wedge$ *dnf-to-bool* γ *ds*)
by(*simp add*: *map-a-and dnf-to-bool-append*)

We use $\exists$ to model the big $\vee$ operation

lemma *listprepend-correct*: *dnf-to-bool* γ (*listprepend as ds*) $\longleftrightarrow$ ($\exists$ *a*$\in$ *set as*. *dnf-to-bool* γ [[*a*]] $\wedge$ *dnf-to-bool* γ *ds*)
apply(*induction as*)
apply(*simp*)
apply(*simp*)
apply(*rename-tac a as*)
apply(*simp add*: *map-a-and cnf-to-bool-append dnf-to-bool-append*)
by *blast*

lemma *listprepend-correct'*: *dnf-to-bool* γ (*listprepend as ds*) $\longleftrightarrow$ (*dnf-to-bool* γ (*map* (λa. [*a*]) *as*) $\wedge$ *dnf-to-bool* γ *ds*)
apply(*induction as*)
apply(*simp*)
apply(*simp*)
apply(*rename-tac a as*)
apply(*simp add*: *map-a-and cnf-to-bool-append dnf-to-bool-append*)
by *blast*

lemma *cnf-invert-singelton*: *cnf-to-bool* γ [*invert a*] $\longleftrightarrow$ $\neg$ *cnf-to-bool* γ [*a*]
by(*cases a*, *simp-all*)

lemma *cnf-singleton-false*: ($\exists$ *a'*$\in$*set as*. $\neg$ *cnf-to-bool* γ [*a'*]) $\longleftrightarrow$ $\neg$ *cnf-to-bool* γ *as*
by(*induction* γ *as rule*: *cnf-to-bool.induct*) (*simp-all*)

fun *dnf-not* :: *'a dnf* $\Rightarrow$ *'a dnf* **where**
dnf-not [] = [[]] |
dnf-not (*ns*#*nss*) = *listprepend* (*map invert ns*) (*dnf-not nss*)

lemma *dnf-not*: *dnf-to-bool* γ (*dnf-not d*) $\longleftrightarrow$ $\neg$ *dnf-to-bool* γ *d*
apply(*induction d*)
apply(*simp-all*)
apply(*simp add*: *listprepend-correct*)
apply(*simp add*: *cnf-invert-singelton cnf-singleton-false*)
done

18.0.2 Optimizing

definition *optimize-dfn* :: *'a dnf* $\Rightarrow$ *'a dnf* **where**
optimize-dfn dnf = *map remdups* (*remdups dnf*)

lemma *dnf-to-bool f* (*optimize-dfn dnf*) = *dnf-to-bool f dnf*
unfolding *optimize-dfn-def*
apply(*rule dnf-to-bool-seteq*)
apply(*simp*)
by (*metis image-cong image-image set-remdups*)

end
theory *Matching-Embeddings*
imports *Semantics-Ternary/Matching-Ternary Matching Semantics-Ternary/Unknown-Match-Tacs*
begin

19 Boolean Matching vs. Ternary Matching

term *Semantics.matches*
term *Matching-Ternary.matches*

The two matching semantics are related. However, due to the ternary

logic, we cannot directly translate one to the other. The problem are *MatchNot* expressions which evaluate to *TernaryUnknown* because *MatchNot TernaryUnknown* and *TernaryUnknown* are semantically equal!

lemma $\exists\, m\ \beta\ \alpha\ a.$ *Matching-Ternary.matches* $(\beta, \alpha)\ m\ a\ p \neq$
Semantics.matches $(\lambda\ atm\ p.$ *case* $\beta\ atm\ p$ *of TernaryTrue* $\Rightarrow$ *True* | *TernaryFalse* $\Rightarrow$ *False* | *TernaryUnknown* $\Rightarrow\ \alpha\ a\ p)\ m\ p$
apply(*rule-tac x=MatchNot* (*Match X*) **in** *exI*) — any *X*
by (*auto split*: *ternaryvalue.split ternaryvalue.split-asm simp add*: *matches-case-ternaryvalue-tuple*)

the *the* in the next definition is always defined

lemma $\forall\, m \in \{m.$ *approx m p* $\neq$ *TernaryUnknown*$\}.$ *ternary-to-bool* (*approx m p*) $\neq$ *None*
by(*simp add*: *ternary-to-bool-None*)

The Boolean and the ternary matcher agree (where the ternary matcher is defined)

definition *matcher-agree-on-exact-matches* :: $('a, 'p)$ *matcher* $\Rightarrow$ $('a \Rightarrow 'p \Rightarrow$ *ternaryvalue*$) \Rightarrow$ *bool* **where**
matcher-agree-on-exact-matches exact approx $\equiv \forall\, p\ m.$ *approx m p* $\neq$ *TernaryUnknown* $\longrightarrow$ *exact m p* $=$ *the* (*ternary-to-bool* (*approx m p*))

We say the Boolean and ternary matchers agree iff they return the same result or the ternary matcher returns *TernaryUnknown*.

lemma *matcher-agree-on-exact-matches exact approx* $\longleftrightarrow$ $(\forall\, p\ m.$ *exact m p* $=$ *the* (*ternary-to-bool* (*approx m p*)) $\vee$ *approx m p* $=$ *TernaryUnknown*)
unfolding *matcher-agree-on-exact-matches-def* **by** *blast*
lemma *matcher-agree-on-exact-matches-alt*:
matcher-agree-on-exact-matches exact approx $\longleftrightarrow$ $(\forall\, p\ m.$ *approx m p* $\neq$ *TernaryUnknown* $\longrightarrow$ *bool-to-ternary* (*exact m p*) $=$ *approx m p*)
unfolding *matcher-agree-on-exact-matches-def*
by (*metis* (*full-types*) *bool-to-ternary.simps*(*1*) *bool-to-ternary.simps*(*2*) *option.sel ternary-to-bool.simps*(*1*)
ternary-to-bool.simps(*2*) *ternaryvalue.exhaust*)

lemma *eval-ternary-Not-TrueD*: *eval-ternary-Not m* $=$ *TernaryTrue* $\Longrightarrow$ *m* $=$ *TernaryFalse*
by (*metis eval-ternary-Not.simps*(*1*) *eval-ternary-idempotence-Not*)

lemma *matches-comply-exact*: *ternary-ternary-eval* (*map-match-tac* β *p m*) $\neq$ *TernaryUnknown* $\Longrightarrow$
matcher-agree-on-exact-matches $\gamma\ \beta \Longrightarrow$
Semantics.matches γ *m p* $=$ *Matching-Ternary.matches* $(\beta, \alpha)\ m\ a\ p$
proof(*unfold matches-case-ternaryvalue-tuple*,*induction m*)
case *Match* **thus** *?case*
by(*simp split*: *ternaryvalue.split add*: *matcher-agree-on-exact-matches-def*)
next
case (*MatchNot m*) **thus** *?case*

```
    apply(simp split: ternaryvalue.split add: matcher-agree-on-exact-matches-def)
     apply(case-tac ternary-ternary-eval (map-match-tac β p m))
      by(simp-all)
  next
  case (MatchAnd m1 m2)
   thus ?case
    apply(case-tac ternary-ternary-eval (map-match-tac β p m1))
     apply(case-tac [!] ternary-ternary-eval (map-match-tac β p m2))
           by(simp-all)
  next
  case MatchAny thus ?case by simp
  qed


lemma matcher-agree-on-exact-matches-gammaE:
  matcher-agree-on-exact-matches γ β ⟹ β X p = TernaryTrue ⟹ γ X p
  apply(simp add: matcher-agree-on-exact-matches-alt)
  apply(erule-tac x=p in allE)
  apply(erule-tac x=X in allE)
  apply(simp add: bool-to-ternary-simps)
  done



lemma in-doubt-allow-allows-Accept: a = Accept ⟹ matcher-agree-on-exact-matches
γ β ⟹
      Semantics.matches γ m p ⟹ Matching-Ternary.matches (β, in-doubt-allow)
m a p
 apply(case-tac ternary-ternary-eval (map-match-tac β p m) ≠ TernaryUnknown)
  using matches-comply-exact apply fast
 apply(simp add: matches-case-ternaryvalue-tuple)
 done

lemma not-exact-match-in-doubt-allow-approx-match: matcher-agree-on-exact-matches
γ β ⟹ a = Accept ∨ a = Reject ∨ a = Drop ⟹
 ¬ Semantics.matches γ m p ⟹
 (a = Accept ∧ Matching-Ternary.matches (β, in-doubt-allow) m a p) ∨ ¬ Match-
ing-Ternary.matches (β, in-doubt-allow) m a p
 apply(case-tac ternary-ternary-eval (map-match-tac β p m) ≠ TernaryUnknown)
  apply(drule(1) matches-comply-exact[where α=in-doubt-allow and a=a])
  apply(rule disjI2)
  apply fast
 apply(simp)
 apply(clarify)
 apply(simp add: matches-case-ternaryvalue-tuple)
 apply(cases a)
       apply(simp-all)
 done
```

lemma *in-doubt-deny-denies-DropReject*: $a = Drop \vee a = Reject \Longrightarrow$ *matcher-agree-on-exact-matches* $\gamma\ \beta \Longrightarrow$
Semantics.matches $\gamma\ m\ p \Longrightarrow$ *Matching-Ternary.matches* (β, *in-doubt-deny*) $m\ a\ p$
apply(*case-tac ternary-ternary-eval* (*map-match-tac* $\beta\ p\ m$) $\neq$ *TernaryUnknown*)
using *matches-comply-exact* **apply** *fast*
apply(*simp*)
apply(*auto simp add*: *matches-case-ternaryvalue-tuple*)
done

lemma *not-exact-match-in-doubt-deny-approx-match*: *matcher-agree-on-exact-matches* $\gamma\ \beta \Longrightarrow a = Accept \vee a = Reject \vee a = Drop \Longrightarrow$
$\neg$ *Semantics.matches* $\gamma\ m\ p \Longrightarrow$
$((a = Drop \vee a = Reject) \wedge$ *Matching-Ternary.matches* (β, *in-doubt-deny*) $m\ a\ p) \vee \neg$ *Matching-Ternary.matches* (β, *in-doubt-deny*) $m\ a\ p$
apply(*case-tac ternary-ternary-eval* (*map-match-tac* $\beta\ p\ m$) $\neq$ *TernaryUnknown*)
apply(*drule*(*1*) *matches-comply-exact*[**where** α=*in-doubt-deny* **and** a=a])
apply(*rule disjI2*)
apply *fast*
apply(*simp*)
apply(*clarify*)
apply(*simp add*: *matches-case-ternaryvalue-tuple*)
apply(*cases a*)
apply(*simp-all*)
done

The ternary primitive matcher can return exactly the result of the Boolean primitive matcher

definition β_{magic} :: ('*a*, '*p*) *matcher* $\Rightarrow$ ('*a* $\Rightarrow$ '*p* $\Rightarrow$ *ternaryvalue*) **where**
$\beta_{magic}\ \gamma \equiv (\lambda\ a\ p.$ *if* $\gamma\ a\ p$ *then TernaryTrue else TernaryFalse*)

lemma *matcher-agree-on-exact-matches* γ ($\beta_{magic}\ \gamma$)
by(*simp add*: *matcher-agree-on-exact-matches-def* β_{magic}-*def*)

lemma β_{magic}-*not-Unknown*: *ternary-ternary-eval* (*map-match-tac* ($\beta_{magic}\ \gamma$) p m) $\neq$ *TernaryUnknown*
proof(*induction m*)
case *MatchNot* **thus** *?case* **using** *eval-ternary-Not-UnknownD* β_{magic}-*def*
by (*simp*) *blast*
case (*MatchAnd m1 m2*) **thus** *?case*
apply(*case-tac ternary-ternary-eval* (*map-match-tac* ($\beta_{magic}\ \gamma$) p *m1*))
apply(*case-tac* [!] *ternary-ternary-eval* (*map-match-tac* ($\beta_{magic}\ \gamma$) p *m2*))
by(*simp-all add*: β_{magic}-*def*)
qed (*simp-all add*: β_{magic}-*def*)

lemma β_{magic}-*matching*: *Matching-Ternary.matches* ((β_{magic} γ), α) *m a p* $\longleftrightarrow$ *Semantics.matches* γ *m p*
proof(*induction m*)
case *Match* **thus** *?case*
by(*simp add*: β_{magic}-*def matches-case-ternaryvalue-tuple*)
case *MatchNot* **thus** *?case*
by(*simp add*: *matches-case-ternaryvalue-tuple* β_{magic}-*not-Unknown split*: *ternaryvalue.split-asm*)
qed (*simp-all add*: *matches-case-ternaryvalue-tuple split*: *ternaryvalue.split ternaryvalue.split-asm*)

end
theory *Fixed-Action*
imports *Semantics-Ternary*
begin

20 Fixed Action

If firewall rules have the same action, we can focus on the matching only.

Applying a rule once or several times makes no difference.

lemma *approximating-bigstep-fun-prepend-replicate*:
n > *0* $\Longrightarrow$ *approximating-bigstep-fun* γ *p* (*r*#*rs*) *Undecided* = *approximating-bigstep-fun* γ *p* ((*replicate n r*)@*rs*) *Undecided*
apply(*induction n*)
apply(*simp*)
apply(*simp*)
apply(*case-tac r*)
apply(*rename-tac m a*)
apply(*simp split*: *action.split*)
by *fastforce*

utility lemmas

context
begin
private lemma *fixedaction-Log*: *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m Log*) *ms*) *Undecided* = *Undecided*
by(*induction ms*, *simp-all*)

private lemma *fixedaction-Empty*:*approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m Empty*) *ms*) *Undecided* = *Undecided*
by(*induction ms*, *simp-all*)

private lemma *helperX1-Log*: *matches* γ *m′ Log p* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*map* ((λ*m*. *Rule m Log*) $\circ$ *MatchAnd m′*) *m2′* @ *rs2*) *Undecided* =

```
      approximating-bigstep-fun γ p rs2 Undecided
  by(induction m2')(simp-all split: action.split)

 private lemma helperX1-Empty: matches γ m' Empty p ⟹
      approximating-bigstep-fun γ p (map ((λm. Rule m Empty) ∘ MatchAnd m')
m2' @ rs2) Undecided =
      approximating-bigstep-fun γ p rs2 Undecided
  by(induction m2')(simp-all split: action.split)

 private lemma helperX3: matches γ m' a p ⟹
      approximating-bigstep-fun γ p (map ((λm. Rule m a) ∘ MatchAnd m') m2'
@ rs2 ) Undecided =
     approximating-bigstep-fun γ p (map (λm. Rule m a) m2' @ rs2) Undecided
 proof(induction m2')
  case Nil thus ?case by simp
  next
  case Cons thus ?case by(cases a) (simp-all add: matches-simps)
 qed

  lemmas fixed-action-simps = fixedaction-Log fixedaction-Empty helperX1-Log
helperX1-Empty helperX3
end

lemma fixedaction-swap:
   approximating-bigstep-fun γ p (map (λm. Rule m a) (m1@m2)) s = approxi-
mating-bigstep-fun γ p (map (λm. Rule m a) (m2@m1)) s
proof(induction s rule: just-show-all-approximating-bigstep-fun-equalities-with-start-Undecided)
case Undecided
 have approximating-bigstep-fun γ p (map (λm. Rule m a) m1 @ map (λm. Rule
m a) m2) Undecided =
      approximating-bigstep-fun γ p (map (λm. Rule m a) m2 @ map (λm. Rule
m a) m1) Undecided
 proof(induction m1)
  case Nil thus ?case by simp
  next
  case (Cons m m1)
   { fix m rs
      have approximating-bigstep-fun γ p ((map (λm. Rule m Log) m)@rs)
Undecided =
       approximating-bigstep-fun γ p rs Undecided
     by(induction m) (simp-all)
   } note Log-helper=this
   { fix m rs
     have approximating-bigstep-fun γ p ((map (λm. Rule m Empty) m)@rs)
Undecided =
       approximating-bigstep-fun γ p rs Undecided
     by(induction m) (simp-all)
   } note Empty-helper=this
```

```
    show ?case
     proof(cases matches γ m a p)
      case True
       thus ?thesis
        proof(induction m2)
         case Nil thus ?case by simp
        next
         case Cons thus ?case
          apply(simp split:action.split action.split-asm)
          using Log-helper Empty-helper by fastforce+
        qed
      next
      case False
       thus ?thesis
       apply(simp)
       apply(simp add: Cons.IH)
       apply(induction m2)
        apply(simp-all)
       apply(simp split:action.split action.split-asm)
       apply fastforce
      done
     qed
   qed
  thus ?thesis using Undecided by simp
qed

corollary fixedaction-reorder: approximating-bigstep-fun γ p (map (λm. Rule m a)
(m1 @ m2 @ m3)) s = approximating-bigstep-fun γ p (map (λm. Rule m a) (m2
@ m1 @ m3)) s
proof(induction s rule: just-show-all-approximating-bigstep-fun-equalities-with-start-Undecided)
case Undecided
have approximating-bigstep-fun γ p (map (λm. Rule m a) (m1 @ m2 @ m3))
Undecided = approximating-bigstep-fun γ p (map (λm. Rule m a) (m2 @ m1 @
m3)) Undecided
  proof(induction m3)
    case Nil thus ?case using fixedaction-swap by fastforce
    next
    case (Cons m3'1 m3)
      have approximating-bigstep-fun γ p (map (λm. Rule m a) ((m3'1 # m3)
@ m1 @ m2)) Undecided = approximating-bigstep-fun γ p (map (λm. Rule m a)
((m3'1 # m3) @ m2 @ m1)) Undecided
       apply(simp)
       apply(cases matches γ m3'1 a p)
        apply(simp split: action.split action.split-asm)
        apply (metis append-assoc fixedaction-swap map-append Cons.IH)
       apply(simp)
       by (metis append-assoc fixedaction-swap map-append Cons.IH)
     hence approximating-bigstep-fun γ p (map (λm. Rule m a) ((m1 @ m2) @
m3'1 # m3)) Undecided = approximating-bigstep-fun γ p (map (λm. Rule m a)
```

```
((m2 @ m1) @ m3'1 # m3)) Undecided
      apply(subst fixedaction-swap)
      apply(subst(2) fixedaction-swap)
      by simp
    thus ?case
      apply(subst append-assoc[symmetric])
      apply(subst append-assoc[symmetric])
      by simp
 qed
 thus ?thesis using Undecided by simp
qed
```

If the actions are equal, the *set* (position and replication independent) of the match expressions can be considered.

```
lemma approximating-bigstep-fun-fixaction-matchseteq: set m1 = set m2 ⟹
      approximating-bigstep-fun γ p (map (λm. Rule m a) m1) s =
     approximating-bigstep-fun γ p (map (λm. Rule m a) m2) s
proof(rule just-show-all-approximating-bigstep-fun-equalities-with-start-Undecided)
 assume m1m2-seteq: set m1 = set m2 and s = Undecided
 from m1m2-seteq have
   approximating-bigstep-fun γ p (map (λm. Rule m a) m1) Undecided =
    approximating-bigstep-fun γ p (map (λm. Rule m a) m2) Undecided
 proof(induction m1 arbitrary: m2)
  case Nil thus ?case by simp
  next
  case (Cons m m1)
   show ?case
    proof (cases m ∈ set m1)
    case True
      from True have set m1 = set (m # m1) by auto
    from Cons.IH[OF ‹set m1 = set (m # m1)›] have approximating-bigstep-fun
γ p (map (λm. Rule m a) (m # m1)) Undecided = approximating-bigstep-fun γ p
(map (λm. Rule m a) (m1)) Undecided ..
      thus ?thesis by (metis Cons.IH Cons.prems ‹set m1 = set (m # m1)›)
    next
    case False
      from False have m ∉ set m1 .
      show ?thesis
      proof (cases m ∉ set m2)
        case True
        from True ‹m ∉ set m1› Cons.prems have set m1 = set m2 by auto
       from Cons.IH[OF this] show ?thesis by (metis Cons.IH Cons.prems ‹set
m1 = set m2›)
      next
      case False
        hence m ∈ set m2 by simp

        have repl-filter-simp: (replicate (length [x←m2 . x = m]) m) = [x←m2 .
x = m]
```

using *replicate-length-filter* [*of m m2*] **by** (*metis* (*mono-tags*, *lifting*) *filter-cong*)

from *Cons.prems* ‹*m* ∉ *set m1*› **have** *set m1* = *set* (*filter* (λ*x*. *x*≠*m*) *m2*) **by** *auto*
from *Cons.IH*[*OF this*] **have** *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) *m1*) *Undecided* = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) [*x*←*m2* . *x* ≠ *m*]) *Undecided* **.**
from *this* **have** *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*m*#*m1*)) *Undecided* = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*m*#[*x*←*m2* . *x* ≠ *m*])) *Undecided*
apply (*simp split*: *action.split*)
by *fast*
also have ... = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) ([*x*←*m2* . *x* = *m*]@[*x*←*m2* . *x* ≠ *m*])) *Undecided*
apply (*simp only*: *list.map*)
apply (*subst approximating-bigstep-fun-prepend-replicate*[**where** *n*=*length* [*x*←*m2* . *x* = *m*]])
apply (*metis* (*full-types*) *False filter-empty-conv neq0-conv repl-filter-simp replicate-0*)
apply (*metis* (*lifting*, *no-types*) *map-append map-replicate repl-filter-simp*)
done
also have ... = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) *m2*) *Undecided*
proof(*induction m2*)
case *Nil* **thus** *?case* **by** *simp*
next
case(*Cons m2'1 m2'*)
have *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) [*x*←*m2'* . *x* = *m*] @ *Rule m2'1 a* # *map* (λ*m*. *Rule m a*) [*x*←*m2'* . *x* ≠ *m*]) *Undecided* =
approximating-bigstep-fun γ *p* (*map* (λ*m*. *Rule m a*) ([*x*←*m2'* . *x* = *m*] @ [*m2'1*] @ [*x*←*m2'* . *x* ≠ *m*])) *Undecided* **by** *fastforce*
also have ... = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) ([*m2'1*] @ [*x*←*m2'* . *x* = *m*] @ [*x*←*m2'* . *x* ≠ *m*])) *Undecided*
using *fixedaction-reorder* **by** *fast*
finally have *XX*: *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) [*x*←*m2'* . *x* = *m*] @ *Rule m2'1 a* # *map* (λ*m*. *Rule m a*) [*x*←*m2'* . *x* ≠ *m*]) *Undecided* =
approximating-bigstep-fun γ *p* (*Rule m2'1 a* # (*map* (λ*m*. *Rule m a*) [*x*←*m2'* . *x* = *m*] @ *map* (λ*m*. *Rule m a*) [*x*←*m2'* . *x* ≠ *m*])) *Undecided*
by *fastforce*
from *Cons* **show** *?case*
apply(*case-tac m2'1* = *m*)
apply(*simp split*: *action.split*)
apply *fast*
apply(*simp del*: *approximating-bigstep-fun.simps*)
apply(*simp only*: *XX*)
apply(*case-tac matches* γ *m2'1 a p*)
apply(*simp*)

```
            apply(simp split: action.split)
            apply(fast)
           apply(simp)
           done
       qed
      finally show ?thesis .
    qed
  qed
 qed
 thus ?thesis using ‹s = Undecided› by simp
qed
```

20.1 *match-list*

Reducing the firewall semantics to short-circuit matching evaluation

```
 fun match-list :: ('a, 'packet) match-tac ⇒ 'a match-expr list ⇒ action ⇒ 'packet
⇒ bool where
  match-list γ [] a p = False |
  match-list γ (m#ms) a p = (if matches γ m a p then True else match-list γ ms
a p)


 lemma match-list-matches: match-list γ ms a p ⟷ (∃ m ∈ set ms. matches γ
m a p)
  by(induction ms, simp-all)

 lemma match-list-True: match-list γ ms a p ⟹ approximating-bigstep-fun γ p
(map (λm. Rule m a) ms) Undecided = (case a of Accept ⇒ Decision FinalAllow
           | Drop ⇒ Decision FinalDeny
           | Reject ⇒ Decision FinalDeny
           | Log ⇒ Undecided
           | Empty ⇒ Undecided
           — unhandled cases
           )
  apply(induction ms)
   apply(simp)
  apply(simp split: if-split-asm action.split)
  apply(simp add: fixed-action-simps)
  done
 lemma match-list-False: ¬ match-list γ ms a p ⟹ approximating-bigstep-fun γ
p (map (λm. Rule m a) ms) Undecided = Undecided
  apply(induction ms)
   apply(simp)
  apply(simp split: if-split-asm action.split)
  done
```

The key idea behind *match-list*: Reducing semantics to match list

```
 lemma match-list-semantics: match-list γ ms1 a p ⟷ match-list γ ms2 a p ⟹
```

approximating-bigstep-fun γ *p* (*map* (λ*m*. *Rule m a*) *ms1*) *s* = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) *ms2*) *s*
apply(*rule just-show-all-approximating-bigstep-fun-equalities-with-start-Undecided*)
apply(*simp*)
apply(*thin-tac s = Undecided*)
apply(*induction ms2*)
apply(*simp*)
apply(*induction ms1*)
apply(*simp*)
apply(*simp split*: *if-split-asm*)
apply(*rename-tac m ms2*)
apply(*simp del*: *approximating-bigstep-fun.simps*)
apply(*simp split*: *if-split-asm del*: *approximating-bigstep-fun.simps*)
apply(*simp split*: *action.split add*: *match-list-True fixed-action-simps*)
apply(*simp*)
done

We can exploit de-morgan to get a disjunction in the match expression!

fun *match-list-to-match-expr* :: *'a match-expr list* $\Rightarrow$ *'a match-expr* **where**
match-list-to-match-expr [] = *MatchNot MatchAny* |
match-list-to-match-expr (*m*#*ms*) = *MatchOr m* (*match-list-to-match-expr ms*)

match-list-to-match-expr constructs a unwieldy *'a match-expr* from a list. The semantics of the resulting match expression is the disjunction of the elements of the list. This is handy because the normal match expressions do not directly support disjunction. Use this function with care because the resulting match expression is very ugly!

lemma *match-list-to-match-expr-disjunction*: *match-list* γ *ms a p* $\longleftrightarrow$ *matches* γ (*match-list-to-match-expr ms*) *a p*
apply(*induction ms rule*: *match-list-to-match-expr.induct*)
apply(*simp add*: *bunch-of-lemmata-about-matches*; *fail*)
apply(*simp add*: *MatchOr*)
done

lemma *match-list-singleton*: *match-list* γ [*m*] *a p* $\longleftrightarrow$ *matches* γ *m a p* **by**(*simp*)

lemma *match-list-append*: *match-list* γ (*m1*@*m2*) *a p* $\longleftrightarrow$ ($\neg$ *match-list* γ *m1 a p* $\longrightarrow$ *match-list* γ *m2 a p*)
by(*induction m1*) *simp*+

lemma *match-list-helper1*: $\neg$ *matches* γ *m2 a p* $\Longrightarrow$ *match-list* γ (*map* (λ*x*. *MatchAnd x m2*) *m1'*) *a p* $\Longrightarrow$ *False*
apply(*induction m1'*)
apply(*simp*; *fail*)
apply(*simp split*:*if-split-asm*)
by(*auto dest*: *matches-dest*)
lemma *match-list-helper2*: $\neg$ *matches* γ *m a p* $\Longrightarrow$ $\neg$ *match-list* γ (*map* (*MatchAnd m*) *m2'*) *a p*
apply(*induction m2'*)

apply(*simp*; *fail*)
apply(*simp split*:*if-split-asm*)
by(*auto dest*: *matches-dest*)
lemma *match-list-helper3*: *matches* γ *m a p* $\Longrightarrow$ *match-list* γ *m2'* *a p* $\Longrightarrow$ *match-list* γ (*map* (*MatchAnd m*) *m2'*) *a p*
apply(*induction m2'*)
apply(*simp*; *fail*)
apply(*simp split*:*if-split-asm*)
by (*simp add*: *matches-simps*)
lemma *match-list-helper4*: $\neg$ *match-list* γ *m2'* *a p* $\Longrightarrow$ $\neg$ *match-list* γ (*map* (*MatchAnd aa*) *m2'*) *a p*
apply(*induction m2'*)
apply(*simp*; *fail*)
apply(*simp split*:*if-split-asm*)
by(*auto dest*: *matches-dest*)
lemma *match-list-helper5*: $\neg$ *match-list* γ *m2'* *a p* $\Longrightarrow$ $\neg$ *match-list* γ (*concat* (*map* (λ*x*. *map* (*MatchAnd x*) *m2'*) *m1'*)) *a p*
apply(*induction m2'*)
apply(*simp add*:*empty-concat*; *fail*)
apply(*simp split*:*if-split-asm*)
apply(*induction m1'*)
apply(*simp*; *fail*)
apply(*simp add*: *match-list-append*)
by(*auto dest*: *matches-dest*)
lemma *match-list-helper6*: $\neg$ *match-list* γ *m1'* *a p* $\Longrightarrow$ $\neg$ *match-list* γ (*concat* (*map* (λ*x*. *map* (*MatchAnd x*) *m2'*) *m1'*)) *a p*
apply(*induction m2'*)
apply(*simp add*:*empty-concat*; *fail*)
apply(*simp split*:*if-split-asm*)
apply(*induction m1'*)
apply(*simp*; *fail*)
apply(*simp add*: *match-list-append split*: *if-split-asm*)
by(*auto dest*: *matches-dest*)

lemmas *match-list-helper* = *match-list-helper1 match-list-helper2 match-list-helper3 match-list-helper4 match-list-helper5 match-list-helper6*
hide-fact *match-list-helper1 match-list-helper2 match-list-helper3 match-list-helper4 match-list-helper5 match-list-helper6*

lemma *match-list-map-And1*: *matches* γ *m1 a p* = *match-list* γ *m1'* *a p* $\Longrightarrow$
matches γ (*MatchAnd m1 m2*) *a p* $\longleftrightarrow$ *match-list* γ (*map* (λ*x*. *MatchAnd x m2*) *m1'*) *a p*
apply(*induction m1'*)
apply(*auto dest*: *matches-dest*; *fail*)[*1*]
apply(*simp split*: *if-split-asm*)
apply *safe*
apply(*simp-all add*: *matches-simps*)
apply(*auto dest*: *match-list-helper*(*1*))[*1*]
by(*auto dest*: *matches-dest*)

lemma *matches-list-And-concat*: *matches* γ *m1 a p* = *match-list* γ *m1'* *a p* $\Longrightarrow$ *matches* γ *m2 a p* = *match-list* γ *m2'* *a p* $\Longrightarrow$
matches γ (*MatchAnd m1 m2*) *a p* $\longleftrightarrow$ *match-list* γ [*MatchAnd x y. x* <− *m1'*, *y* <− *m2'*] *a p*
apply(*induction m1'*)
apply(*auto dest*: *matches-dest*; *fail*)[*1*]
apply(*simp split*: *if-split-asm*)
prefer *2*
apply(*simp add*: *match-list-append*)
apply(*subgoal-tac* ¬ *match-list* γ (*map* (*MatchAnd aa*) *m2'*) *a p*)
apply(*simp*; *fail*)
apply *safe*
apply(*simp-all add*: *matches-simps match-list-append match-list-helper*)
done

lemma *match-list-concat*: *match-list* γ (*concat lss*) *a p* $\longleftrightarrow$ (∃ *ls* ∈ *set lss*. *match-list* γ *ls a p*)
apply(*induction lss*)
apply(*simp*; *fail*)
by(*auto simp add*: *match-list-append*)

lemma *fixedaction-wf-ruleset*: *wf-ruleset* γ *p* (*map* (λ*m*. *Rule m a*) *ms*) $\longleftrightarrow$
¬ *match-list* γ *ms a p* ∨ ¬ (∃ *chain*. *a* = *Call chain*) ∧ *a* ≠ *Return* ∧ ¬ (∃ *chain*. *a* = *Goto chain*) ∧ *a* ≠ *Unknown*
proof −
have *helper*: ⋀*a b c*. *a* $\longleftrightarrow$ *c* $\Longrightarrow$ (*a* $\longrightarrow$ *b*) = (*c* $\longrightarrow$ *b*) **by** *fast*
show *?thesis*
apply(*simp add*: *wf-ruleset-def*)
apply(*rule helper*)
apply(*induction ms*)
apply(*simp*; *fail*)
apply(*simp*)
done
qed

lemma *wf-ruleset-singleton*: *wf-ruleset* γ *p* [*Rule m a*] $\longleftrightarrow$ ¬ *matches* γ *m a p* ∨ ¬ (∃ *chain*. *a* = *Call chain*) ∧ *a* ≠ *Return* ∧ ¬ (∃ *chain*. *a* = *Goto chain*) ∧ *a* ≠ *Unknown*
by(*simp add*: *wf-ruleset-def*)

end
theory *Normalized-Matches*
imports *Fixed-Action*
begin

21 Normalized (DNF) matches

simplify a match expression. The output is a list of match exprissions, the semantics is $\vee$ of the list elements.

fun *normalize-match* :: *'a match-expr* $\Rightarrow$ *'a match-expr list* **where**
normalize-match (*MatchAny*) = [*MatchAny*] |
normalize-match (*Match m*) = [*Match m*] |
normalize-match (*MatchAnd m1 m2*) = [*MatchAnd x y*. *x* <− *normalize-match m1*, *y* <− *normalize-match m2*] |
normalize-match (*MatchNot* (*MatchAnd m1 m2*)) = *normalize-match* (*MatchNot m1*) @ *normalize-match* (*MatchNot m2*) |
normalize-match (*MatchNot* (*MatchNot m*)) = *normalize-match m* |
normalize-match (*MatchNot* (*MatchAny*)) = [] |
normalize-match (*MatchNot* (*Match m*)) = [*MatchNot* (*Match m*)]

lemma *normalize-match-not-matcheq-matchNone*: $\forall\, m' \in$ *set* (*normalize-match m*). $\neg$ *matcheq-matchNone m'*
proof(*induction m rule*: *normalize-match.induct*)
case *4* **thus** *?case* **by** (*simp*) *blast*
qed(*simp-all*)

lemma *normalize-match-empty-iff-matcheq-matchNone*: *normalize-match m* = [] $\longleftrightarrow$ *matcheq-matchNone m*
proof(*induction m rule*: *normalize-match.induct*)
case *3* **thus** *?case* **by** (*simp*) *fastforce*
qed(*simp-all*)

lemma *match-list-normalize-match*: *match-list* γ [*m*] *a p* $\longleftrightarrow$ *match-list* γ (*normalize-match m*) *a p*
proof(*induction m rule*:*normalize-match.induct*)
case *1* **thus** *?case* **by**(*simp add*: *match-list-singleton*)
next
case *2* **thus** *?case* **by**(*simp add*: *match-list-singleton*)
next
case (*3 m1 m2*) **thus** *?case*
apply(*simp-all add*: *match-list-singleton del*: *match-list.simps*(*2*))
apply(*case-tac matches* γ *m1 a p*)
apply(*rule matches-list-And-concat*)
apply(*simp*)
apply(*case-tac* (*normalize-match m1*))
apply *simp*
apply (*auto*)[*1*]
apply(*simp add*: *bunch-of-lemmata-about-matches match-list-helper*)
done
next
case *4* **thus** *?case*
apply(*simp-all add*: *match-list-singleton del*: *match-list.simps*(*2*))
apply(*simp add*: *match-list-append*)

```
    apply(safe)
      apply(simp-all add: matches-DeMorgan)
    done
  next
  case 5 thus ?case
    by(simp add: match-list-singleton bunch-of-lemmata-about-matches)
  next
  case 6 thus ?case
    by(simp add: match-list-singleton bunch-of-lemmata-about-matches)
  next
  case 7 thus ?case by(simp add: match-list-singleton)
qed

thm match-list-normalize-match[simplified match-list-singleton]


theorem normalize-match-correct: approximating-bigstep-fun γ p (map (λm. Rule
m a) (normalize-match m)) s = approximating-bigstep-fun γ p [Rule m a] s
apply(rule match-list-semantics[of - - - - [m], simplified])
using match-list-normalize-match by fastforce


lemma normalize-match-empty: normalize-match m = [] ⟹ ¬ matches γ m a p
  proof(induction m rule: normalize-match.induct)
  case 3 thus ?case by(fastforce dest: matches-dest)
  next
 case 4 thus ?case using match-list-normalize-match by (simp add: matches-DeMorgan)
  next
  case 5 thus ?case using matches-not-idem by fastforce
  next
  case 6 thus ?case by(simp add: bunch-of-lemmata-about-matches)
  qed(simp-all)


lemma matches-to-match-list-normalize: matches γ m a p = match-list γ (normalize-match
m) a p
  using match-list-normalize-match[simplified match-list-singleton] .

lemma wf-ruleset-normalize-match: wf-ruleset γ p [(Rule m a)] ⟹ wf-ruleset γ
p (map (λm. Rule m a) (normalize-match m))
proof(induction m rule: normalize-match.induct)
  case 1 thus ?case by simp
  next
  case 2 thus ?case by simp
  next
 case 3 thus ?case by(simp add: fixedaction-wf-ruleset wf-ruleset-singleton matches-to-match-list-normalize)
  next
  case 4 thus ?case
    apply(simp add: wf-ruleset-append)
```

apply(*simp add*: *fixedaction-wf-ruleset*)
apply(*unfold wf-ruleset-singleton*)
apply(*safe*)
apply(*simp-all add*: *matches-to-match-list-normalize*)
apply(*simp-all add*: *match-list-append*)
done
next
case *5* **thus** *?case* **by**(*simp add*: *wf-ruleset-singleton matches-to-match-list-normalize*)
next
case *6* **thus** *?case* **by**(*simp add*: *wf-ruleset-def*)
next
case *7* **thus** *?case* **by**(*simp-all add*: *wf-ruleset-append*)
qed

lemma *normalize-match-wf-ruleset*: *wf-ruleset* γ *p* (*map* (λ*m*. *Rule m a*) (*normalize-match m*)) $\Longrightarrow$ *wf-ruleset* γ *p* [*Rule m a*]
proof(*induction m rule*: *normalize-match.induct*)
case *1* **thus** *?case* **by** *simp*
next
case *2* **thus** *?case* **by** *simp*
next
case *3* **thus** *?case* **by**(*simp add*: *fixedaction-wf-ruleset wf-ruleset-singleton matches-to-match-list-normalize*)
next
case *4* **thus** *?case*
apply(*simp add*: *wf-ruleset-append*)
apply(*simp add*: *fixedaction-wf-ruleset*)
apply(*unfold wf-ruleset-singleton*)
apply(*safe*)
apply(*simp-all add*: *matches-to-match-list-normalize*)
apply(*simp-all add*: *match-list-append*)
done
next
case *5* **thus** *?case*
unfolding *wf-ruleset-singleton* **by**(*simp add*: *matches-to-match-list-normalize*)
next
case *6* **thus** *?case* **unfolding** *wf-ruleset-singleton* **by**(*simp add*: *bunch-of-lemmata-about-matches*)
next
case *7* **thus** *?case* **by**(*simp add*: *wf-ruleset-append*)
qed

lemma *good-ruleset-normalize-match*: *good-ruleset* [(*Rule m a*)] $\Longrightarrow$ *good-ruleset* (*map* (λ*m*. *Rule m a*) (*normalize-match m*))
by(*simp add*: *good-ruleset-def*)

22 Normalizing rules instead of only match expressions

fun *normalize-rules* :: (*'a match-expr* ⇒ *'a match-expr list*) ⇒ *'a rule list* ⇒ *'a rule list* **where**
normalize-rules - [] = [] |
normalize-rules f ((*Rule m a*)#*rs*) = (*map* (λ*m*. *Rule m a*) (*f m*))@(*normalize-rules f rs*)

lemma *normalize-rules-singleton*: *normalize-rules f* [*Rule m a*] = *map* (λ*m*. *Rule m a*) (*f m*) **by**(*simp*)
lemma *normalize-rules-fst*: (*normalize-rules f* (*r* # *rs*)) = (*normalize-rules f* [*r*]) @ (*normalize-rules f rs*)
by(*cases r*) (*simp*)

lemma *normalize-rules-concat-map*:
normalize-rules f rs = *concat* (*map* (λ*r*. *map* (λ*m*. *Rule m* (*get-action r*)) (*f* (*get-match r*))) *rs*)
apply(*induction rs*)
apply(*simp-all*)
apply(*rename-tac r rs*, *case-tac r*)
apply(*simp*)
done

lemma *good-ruleset-normalize-rules*: *good-ruleset rs* ⟹ *good-ruleset* (*normalize-rules f rs*)
proof(*induction rs*)
case *Nil* **thus** *?case* **by** (*simp*)
next
case(*Cons r rs*)
from *Cons* **have** *IH*: *good-ruleset* (*normalize-rules f rs*) **using** *good-ruleset-tail* **by** *blast*
from *Cons.prems* **have** *good-ruleset* [*r*] **using** *good-ruleset-fst* **by** *fast*
hence *good-ruleset* (*normalize-rules f* [*r*]) **by**(*cases r*) (*simp add*: *good-ruleset-alt*)
with *IH good-ruleset-append* **have** *good-ruleset* (*normalize-rules f* [*r*] @ *normalize-rules f rs*) **by** *blast*
thus *?case* **using** *normalize-rules-fst* **by** *metis*
qed

lemma *simple-ruleset-normalize-rules*: *simple-ruleset rs* ⟹ *simple-ruleset* (*normalize-rules f rs*)
proof(*induction rs*)
case *Nil* **thus** *?case* **by** (*simp*)
next
case(*Cons r rs*)
from *Cons* **have** *IH*: *simple-ruleset* (*normalize-rules f rs*) **using** *simple-ruleset-tail* **by** *blast*
from *Cons.prems* **have** *simple-ruleset* [*r*] **using** *simple-ruleset-append* **by** *fastforce*

hence *simple-ruleset* (*normalize-rules f* [*r*]) **by**(*cases r*) (*simp add*: *simple-ruleset-def*)
with *IH simple-ruleset-append* **have** *simple-ruleset* (*normalize-rules f* [*r*] @ *normalize-rules f rs*) **by** *blast*
thus *?case* **using** *normalize-rules-fst* **by** *metis*
qed

lemma *normalize-rules-match-list-semantics-3*:
assumes ∀ *m a*. *P m* ⟶ *match-list γ* (*f m*) *a p* = *matches γ m a p*
and *simple-ruleset rs*
and *P*: ∀ *r* ∈ *set rs*. *P* (*get-match r*)
shows *approximating-bigstep-fun γ p* (*normalize-rules f rs*) *s* = *approximating-bigstep-fun γ p rs s*
proof −
have *assm-1*: ∀ *r*∈*set rs*. *match-list γ* (*f* (*get-match r*)) (*get-action r*) *p* = *matches γ* (*get-match r*) (*get-action r*) *p* **using** *P assms*(*1*) **by** *blast*
{ fix *r s*
assume *r* ∈ *set rs*
with *assm-1* **have** *match-list γ* (*f* (*get-match r*)) (*get-action r*) *p* ⟷ *match-list γ* [(*get-match r*)] (*get-action r*) *p* **by** *simp*
with *match-list-semantics*[*of γ f* (*get-match r*) (*get-action r*) *p* [(*get-match r*)]] **have**
approximating-bigstep-fun γ p (*map* (λ*m*. *Rule m* (*get-action r*)) (*f* (*get-match r*))) *s* =
approximating-bigstep-fun γ p [*Rule* (*get-match r*) (*get-action r*)] *s* **by** *simp*
hence (*approximating-bigstep-fun γ p* (*normalize-rules f* [*r*]) *s*) = *approximating-bigstep-fun γ p* [*r*] *s*
by(*cases r*) (*simp*)
}

with *assms* **show** *?thesis*
proof(*induction rs arbitrary*: *s*)
case *Nil* **thus** *?case* **by** (*simp*)
next
case (*Cons r rs*)
from *Cons.prems* **have** *simple-ruleset* [*r*] **by**(*simp add*: *simple-ruleset-def*)
with *simple-imp-good-ruleset good-imp-wf-ruleset* **have** *wf-r*: *wf-ruleset γ p* [*r*] **by** *fast*

from ‹*simple-ruleset* [*r*]› *simple-imp-good-ruleset good-imp-wf-ruleset* **have** *wf-r*:
wf-ruleset γ p [*r*] **by** *fast*
from *simple-ruleset-normalize-rules*[*OF* ‹*simple-ruleset* [*r*]›] **have** *simple-ruleset* (*normalize-rules f* [*r*])
by(*simp*)
with *simple-imp-good-ruleset good-imp-wf-ruleset* **have** *wf-nr*: *wf-ruleset γ*

p (*normalize-rules f* [*r*]) **by** *fast*

from *Cons* **have** *IH*: $\bigwedge s$. *approximating-bigstep-fun* γ *p* (*normalize-rules f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
using *simple-ruleset-tail* **by** *force*

from *Cons* **have** *a*: $\bigwedge s$. *approximating-bigstep-fun* γ *p* (*normalize-rules f* [*r*]) *s* = *approximating-bigstep-fun* γ *p* [*r*] *s* **by** *simp*

show *?case*
apply(*subst normalize-rules-fst*)
apply(*simp add*: *approximating-bigstep-fun-seq-wf*[*OF wf-nr*])
apply(*subst approximating-bigstep-fun-seq-wf*[*OF wf-r*, *simplified*])
apply(*simp add*: *a*)
apply(*simp add*: *IH*)
done
qed
qed

corollary *normalize-rules-match-list-semantics*:
($\forall$ *m a*. *match-list* γ (*f m*) *a p* = *matches* γ *m a p*) $\Longrightarrow$ *simple-ruleset rs* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*normalize-rules f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
by(*rule normalize-rules-match-list-semantics-3*[**where** *P*=λ-. *True*]) *simp-all*

lemma *in-normalized-matches*: *ls* $\in$ *set* (*normalize-match m*) $\wedge$ *matches* γ *ls a p* $\Longrightarrow$ *matches* γ *m a p*
by (*meson match-list-matches matches-to-match-list-normalize*)

applying a function (with a prerequisite Q) to all rules

lemma *normalize-rules-property*:
assumes $\forall$ *r* $\in$ *set rs*. *P* (*get-match r*)
and $\forall$ *m*. *P m* $\longrightarrow$ ($\forall$ *m'* $\in$ *set* (*f m*). *Q m'*)
shows $\forall$ *r* $\in$ *set* (*normalize-rules f rs*). *Q* (*get-match r*)
proof
fix *r'* **assume** *a*: *r'* $\in$ *set* (*normalize-rules f rs*)
from *a assms* **show** *Q* (*get-match r'*)
proof(*induction rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
{
assume *r'* $\in$ *set* (*normalize-rules f rs*)
from *Cons.IH this* **have** *Q* (*get-match r'*) **using** *Cons.prems*(*2*) *Cons.prems*(*3*) **by** *fastforce*
} **note** *1*=*this*
{
assume *r'* $\in$ *set* (*normalize-rules f* [*r*])
hence *a*: (*get-match r'*) $\in$ *set* (*f* (*get-match r*)) **by**(*cases r*) (*auto*)

```
      with Cons.prems(2) Cons.prems(3) have ∀ m'∈set (f (get-match r)). Q m'
by auto
       with a have Q (get-match r') by blast
     } note 2=this
       from Cons.prems(1) have r' ∈ set (normalize-rules f [r]) ∨ r' ∈ set
(normalize-rules f rs)
       by(subst(asm) normalize-rules-fst) auto
     with 1 2 show ?case
       by(elim disjE)(simp)
   qed
 qed
```

If a function f preserves some property of the match expressions, then this property is preserved when applying *normalize-rules*

```
 lemma normalize-rules-preserves: assumes ∀ r ∈ set rs. P (get-match r)
    and ∀ m. P m ⟶ (∀ m' ∈ set (f m). P m')
  shows ∀ r ∈ set (normalize-rules f rs). P (get-match r)
  using normalize-rules-property[OF assms(1) assms(2)] by simp

fun normalize-rules-dnf :: 'a rule list ⇒ 'a rule list where
  normalize-rules-dnf [] = [] |
  normalize-rules-dnf ((Rule m a)#rs) = (map (λm. Rule m a) (normalize-match
m))@(normalize-rules-dnf rs)

lemma normalize-rules-dnf-append: normalize-rules-dnf (rs1@rs2) = normalize-rules-dnf
rs1 @ normalize-rules-dnf rs2
  proof(induction rs1 rule: normalize-rules-dnf.induct)
  qed(simp-all)

lemma normalize-rules-dnf-def2: normalize-rules-dnf = normalize-rules normal-
ize-match
  proof(simp add: fun-eq-iff, intro allI)
   fix x::'a rule list show normalize-rules-dnf x = normalize-rules normalize-match
x
    proof(induction x)
    case (Cons r rs) thus ?case by (cases r) simp
    qed(simp)
  qed

lemma wf-ruleset-normalize-rules-dnf: wf-ruleset γ p rs ⟹ wf-ruleset γ p (normalize-rules-dnf
rs)
  proof(induction rs)
  case Nil thus ?case by simp
  next
  case(Cons r rs)
    from Cons have IH: wf-ruleset γ p (normalize-rules-dnf rs) by(auto dest:
wf-rulesetD)
   from Cons.prems have wf-ruleset γ p [r] by(auto dest: wf-rulesetD)
  hence wf-ruleset γ p (normalize-rules-dnf [r]) using wf-ruleset-normalize-match
```

by(*cases r*) *simp*
with *IH wf-ruleset-append* **have** *wf-ruleset* γ *p* (*normalize-rules-dnf* [*r*] @ *normalize-rules-dnf rs*) **by** *fast*
thus *?case* **using** *normalize-rules-dnf-def2 normalize-rules-fst* **by** *metis*
qed

lemma *good-ruleset-normalize-rules-dnf*: *good-ruleset rs* $\Longrightarrow$ *good-ruleset* (*normalize-rules-dnf rs*)
using *normalize-rules-dnf-def2 good-ruleset-normalize-rules* **by** *metis*

lemma *simple-ruleset-normalize-rules-dnf*: *simple-ruleset rs* $\Longrightarrow$ *simple-ruleset* (*normalize-rules-dnf rs*)
using *normalize-rules-dnf-def2 simple-ruleset-normalize-rules* **by** *metis*

lemma *simple-ruleset rs* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*normalize-rules-dnf rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
unfolding *normalize-rules-dnf-def2*
apply(*rule normalize-rules-match-list-semantics*)
apply (*metis matches-to-match-list-normalize*)
by *simp*

lemma *normalize-rules-dnf-correct*: *wf-ruleset* γ *p rs* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*normalize-rules-dnf rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof(*induction rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
show *?case*
proof(*induction s rule*: *just-show-all-approximating-bigstep-fun-equalities-with-start-Undecided*)
case *Undecided*
from *Cons wf-rulesetD*(*2*) **have** *IH*: *approximating-bigstep-fun* γ *p* (*normalize-rules-dnf rs*) *s* = *approximating-bigstep-fun* γ *p rs s* **by** *fast*
from *Cons.prems* **have** *wf-ruleset* γ *p* [*r*] **and** *wf-ruleset* γ *p* (*normalize-rules-dnf* [*r*])
by(*auto dest*: *wf-rulesetD simp*: *wf-ruleset-normalize-rules-dnf*)
with *IH Undecided* **have**
approximating-bigstep-fun γ *p* (*normalize-rules-dnf rs*) (*approximating-bigstep-fun* γ *p* (*normalize-rules-dnf* [*r*]) *Undecided*) = *approximating-bigstep-fun* γ *p* (*r* # *rs*) *Undecided*
apply(*cases r*, *rename-tac m a*)
apply(*simp*)
apply(*case-tac a*)
apply(*simp-all add*: *normalize-match-correct Decision-approximating-bigstep-fun wf-ruleset-singleton*)
done

hence *approximating-bigstep-fun* γ *p* (*normalize-rules-dnf* [*r*] @ *normalize-rules-dnf rs*) *s* = *approximating-bigstep-fun* γ *p* (*r* # *rs*) *s*
using *Undecided* ‹*wf-ruleset* γ *p* [*r*]› ‹*wf-ruleset* γ *p* (*normalize-rules-dnf* [*r*])›

by(*simp add*: *approximating-bigstep-fun-seq-wf*)
thus *?thesis* **using** *normalize-rules-fst normalize-rules-dnf-def2* **by** *metis*
qed
qed

fun *normalized-nnf-match* :: *'a match-expr* ⇒ *bool* **where**
normalized-nnf-match MatchAny = *True* |
normalized-nnf-match (*Match* -) = *True* |
normalized-nnf-match (*MatchNot* (*Match* -)) = *True* |
normalized-nnf-match (*MatchAnd m1 m2*) = ((*normalized-nnf-match m1*) ∧ (*normalized-nnf-match m2*)) |
normalized-nnf-match - = *False*

Essentially, *normalized-nnf-match* checks for a negation normal form: Only AND is at toplevel, negation only occurs in front of literals. Since *'a match-expr* does not support OR, the result is in conjunction normal form. Applying *normalize-match*, the reuslt is a list. Essentially, this is the disjunctive normal form.

lemma *normalize-match-already-normalized*: *normalized-nnf-match m* ⟹ *normalize-match m* = [*m*]
by(*induction m rule*: *normalize-match.induct*) (*simp*)+

lemma *normalized-nnf-match-normalize-match*: ∀ *m'* ∈ *set* (*normalize-match m*). *normalized-nnf-match m'*
proof(*induction m arbitrary*: *rule*: *normalize-match.induct*)
case *4* **thus** *?case* **by** *fastforce*
qed (*simp-all*)

lemma *normalized-nnf-match-MatchNot-D*: *normalized-nnf-match* (*MatchNot m*) ⟹ *normalized-nnf-match m*
by(*induction m*) (*simp-all*)

Example

lemma *normalize-match* (*MatchNot* (*MatchAnd* (*Match ip-src*) (*Match tcp*))) = [*MatchNot* (*Match ip-src*), *MatchNot* (*Match tcp*)] **by** *simp*

22.1 Functions which preserve *normalized-nnf-match*

lemma *optimize-matches-option-normalized-nnf-match*: (⋀ *r*. *r* ∈ *set rs* ⟹ *normalized-nnf-match* (*get-match r*)) ⟹
(⋀*m m'*. *normalized-nnf-match m* ⟹ *f m* = *Some m'* ⟹ *normalized-nnf-match m'*) ⟹
∀ *r* ∈ *set* (*optimize-matches-option f rs*). *normalized-nnf-match* (*get-match r*)

proof(*induction rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
from *Cons.IH Cons.prems* **have** *IH*: ∀ *r*∈*set* (*optimize-matches-option f rs*). *normalized-nnf-match* (*get-match r*) **by** *simp*
from *Cons.prems* **have** ∀ *r*∈*set* (*optimize-matches-option f* [*r*]). *normalized-nnf-match* (*get-match r*)
apply(*cases r*)
apply(*simp split*: *option.split*)
by *force*
with *IH* **show** *?case* **by**(*cases r*, *simp split*: *option.split-asm*)
qed

lemma *optimize-matches-normalized-nnf-match*: ⟦∀ *r* ∈ *set rs*. *normalized-nnf-match* (*get-match r*); ∀ *m*. *normalized-nnf-match m* ⟶ *normalized-nnf-match* (*f m*) ⟧ ⟹
∀ *r* ∈ *set* (*optimize-matches f rs*). *normalized-nnf-match* (*get-match r*)
unfolding *optimize-matches-def*
apply(*rule optimize-matches-option-normalized-nnf-match*)
apply(*simp*; *fail*)
apply(*simp split*: *if-split-asm*)
by *blast*

lemma *normalize-rules-dnf-normalized-nnf-match*: ∀ *x* ∈ *set* (*normalize-rules-dnf rs*). *normalized-nnf-match* (*get-match x*)
proof(*induction rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*) **thus** *?case* **using** *normalized-nnf-match-normalize-match* **by**(*cases r*) *fastforce*
qed

end
theory *Negation-Type-Matching*
imports *../Common/Negation-Type Matching-Ternary ../Datatype-Selectors Normalized-Matches*
begin

23 Negation Type Matching

Transform a *'a negation-type list* to a *'a match-expr* via conjunction.

fun *alist-and* :: *'a negation-type list* ⇒ *'a match-expr* **where**
alist-and [] = *MatchAny* |
alist-and ((*Pos e*)#*es*) = *MatchAnd* (*Match e*) (*alist-and es*) |
alist-and ((*Neg e*)#*es*) = *MatchAnd* (*MatchNot* (*Match e*)) (*alist-and es*)

lemma *normalized-nnf-match-alist-and*: *normalized-nnf-match* (*alist-and as*)
by(*induction as rule*: *alist-and.induct*) *simp-all*

lemma *alist-and-append*: *matches* γ (*alist-and* (*l1* @ *l2*)) *a p* $\longleftrightarrow$ *matches* γ (*MatchAnd* (*alist-and l1*) (*alist-and l2*)) *a p*
proof(*induction l1*)
case *Nil* **thus** *?case* **by** (*simp add*: *bunch-of-lemmata-about-matches*)
next
case (*Cons l l1*) **thus** *?case* **by** (*cases l*) (*simp-all add*: *bunch-of-lemmata-about-matches*)
qed

This version of *alist-and* avoids the trailing *MatchAny*. Only intended for code.

fun *alist-and'* :: *'a negation-type list* $\Rightarrow$ *'a match-expr* **where**
alist-and' [] = *MatchAny* |
alist-and' [*Pos e*] = *Match e* |
alist-and' [*Neg e*] = *MatchNot* (*Match e*)|
alist-and' ((*Pos e*)#*es*) = *MatchAnd* (*Match e*) (*alist-and' es*) |
alist-and' ((*Neg e*)#*es*) = *MatchAnd* (*MatchNot* (*Match e*)) (*alist-and' es*)

lemma *alist-and'*: *matches* (γ, α) (*alist-and' as*) = *matches* (γ, α) (*alist-and as*)
by(*induction as rule*: *alist-and'.induct*) (*simp-all add*: *bunch-of-lemmata-about-matches*)

lemma *normalized-nnf-match-alist-and'*: *normalized-nnf-match* (*alist-and' as*)
by(*induction as rule*: *alist-and'.induct*) *simp-all*

lemma *matches-alist-and-alist-and'*:
matches γ (*alist-and' ls*) *a p* $\longleftrightarrow$ *matches* γ (*alist-and ls*) *a p*
apply(*induction ls rule*: *alist-and'.induct*)
by(*simp add*: *bunch-of-lemmata-about-matches*)+

lemma *alist-and'-append*: *matches* γ (*alist-and'* (*l1* @ *l2*)) *a p* $\longleftrightarrow$ *matches* γ (*MatchAnd* (*alist-and' l1*) (*alist-and' l2*)) *a p*
proof(*induction l1*)
case *Nil* **thus** *?case* **by** (*simp add*: *bunch-of-lemmata-about-matches*)
next
case (*Cons l l1*) **thus** *?case*
apply (*cases l*)
by(*simp-all add*: *matches-alist-and-alist-and' bunch-of-lemmata-about-matches*)
qed

lemma *alist-and-NegPos-map-getNeg-getPos-matches*:
(∀ *m*∈*set* (*getNeg spts*). *matches* γ (*MatchNot* (*Match* (*C m*))) *a p*) ∧
(∀ *m*∈*set* (*getPos spts*). *matches* γ (*Match* (*C m*)) *a p*)
$\longleftrightarrow$
matches γ (*alist-and* (*NegPos-map C spts*)) *a p*
proof(*induction spts rule*: *alist-and.induct*)
qed(*auto simp add*: *bunch-of-lemmata-about-matches*)

fun *negation-type-to-match-expr-f* :: (*'a* ⇒ *'b*) ⇒ *'a negation-type* ⇒ *'b match-expr*
where
negation-type-to-match-expr-f f (*Pos a*) = *Match* (*f a*) |
negation-type-to-match-expr-f f (*Neg a*) = *MatchNot* (*Match* (*f a*))

lemma *alist-and-negation-type-to-match-expr-f-matches*:
matches γ (*alist-and* (*NegPos-map C spts*)) *a p* ⟷
(∀ *m*∈*set spts*. *matches* γ (*negation-type-to-match-expr-f C m*) *a p*)
proof(*induction spts rule*: *alist-and.induct*)
qed(*auto simp add*: *bunch-of-lemmata-about-matches*)

definition *negation-type-to-match-expr* :: *'a negation-type* ⇒ *'a match-expr* **where**
negation-type-to-match-expr m ≡ *negation-type-to-match-expr-f id m*

lemma *negation-type-to-match-expr-simps*:
negation-type-to-match-expr (*Pos e*) = (*Match e*)
negation-type-to-match-expr (*Neg e*) = (*MatchNot* (*Match e*))
by(*simp-all add*: *negation-type-to-match-expr-def*)

lemma *alist-and-negation-type-to-match-expr*: *alist-and* (*n*#*es*) = *MatchAnd* (*negation-type-to-match-expr n*) (*alist-and es*)
by(*cases n*, *simp-all add*: *negation-type-to-match-expr-simps*)

fun *to-negation-type-nnf* :: *'a match-expr* ⇒ *'a negation-type list* **where**
to-negation-type-nnf MatchAny = [] |
to-negation-type-nnf (*Match a*) = [*Pos a*] |
to-negation-type-nnf (*MatchNot* (*Match a*)) = [*Neg a*] |
to-negation-type-nnf (*MatchAnd a b*) = (*to-negation-type-nnf a*) @ (*to-negation-type-nnf b*) |
to-negation-type-nnf - = *undefined*

lemma *normalized-nnf-match m* ⟹ *matches* γ (*alist-and* (*to-negation-type-nnf m*)) *a p* = *matches* γ *m a p*
proof(*induction m rule*: *to-negation-type-nnf.induct*)
qed(*simp-all add*: *bunch-of-lemmata-about-matches alist-and-append*)

Isolating the matching semantics

fun *nt-match-list* :: (*'a*, *'packet*) *match-tac* ⇒ *action* ⇒ *'packet* ⇒ *'a negation-type list* ⇒ *bool* **where**
nt-match-list - - - [] = *True* |
nt-match-list γ *a p* ((*Pos x*)#*xs*) ⟷ *matches* γ (*Match x*) *a p* ∧ *nt-match-list* γ *a p xs* |
nt-match-list γ *a p* ((*Neg x*)#*xs*) ⟷ *matches* γ (*MatchNot* (*Match x*)) *a p* ∧ *nt-match-list* γ *a p xs*

lemma *nt-match-list-matches*: *nt-match-list* γ *a p l* $\longleftrightarrow$ *matches* γ (*alist-and l*) *a p*
apply(*induction l rule*: *alist-and.induct*)
apply(*case-tac* [!] γ)
apply(*simp-all add*: *bunch-of-lemmata-about-matches*)
done

lemma *nt-match-list-simp*: *nt-match-list* γ *a p ms* $\longleftrightarrow$
$(\forall\, m \in set\ (getPos\ ms).\ matches\ \gamma\ (Match\ m)\ a\ p) \wedge (\forall\, m \in set\ (getNeg\ ms).\ matches\ \gamma\ (MatchNot\ (Match\ m))\ a\ p)$
proof(*induction* γ *a p ms rule*: *nt-match-list.induct*)
case *3* **thus** *?case* **by** *fastforce*
qed(*simp-all*)

lemma *matches-alist-and*: *matches* γ (*alist-and l*) *a p* $\longleftrightarrow$ $(\forall\, m \in set\ (getPos\ l).\ matches\ \gamma\ (Match\ m)\ a\ p) \wedge (\forall\, m \in set\ (getNeg\ l).\ matches\ \gamma\ (MatchNot\ (Match\ m))\ a\ p)$
using *nt-match-list-matches nt-match-list-simp* **by** *fast*

end
theory *Primitive-Normalization*
imports *Negation-Type-Matching*
begin

24 Primitive Normalization

24.1 Normalized Primitives

Test if a *disc* is in the match expression. For example, it call tell whether there are some matches for *Src ip*.

fun *has-disc* :: $('a \Rightarrow bool) \Rightarrow 'a\ match\text{-}expr \Rightarrow bool$ **where**
has-disc - MatchAny = False |
has-disc disc (*Match a*) = *disc a* |
has-disc disc (*MatchNot m*) = *has-disc disc m* |
has-disc disc (*MatchAnd m1 m2*) = (*has-disc disc m1* $\vee$ *has-disc disc m2*)

fun *has-disc-negated* :: $('a \Rightarrow bool) \Rightarrow bool \Rightarrow 'a\ match\text{-}expr \Rightarrow bool$ **where**
has-disc-negated - - MatchAny = False |
has-disc-negated disc neg (*Match a*) = (*if disc a then neg else False*) |
has-disc-negated disc neg (*MatchNot m*) = *has-disc-negated disc* ($\neg$ *neg*) *m* |
has-disc-negated disc neg (*MatchAnd m1 m2*) = (*has-disc-negated disc neg m1* $\vee$ *has-disc-negated disc neg m2*)

lemma $\neg$ *has-disc-negated* (λx::*nat*. *x* = *0*) *False* (*MatchAnd* (*Match 0*) (*MatchNot*

(*Match 1*))) **by** *eval*
lemma *has-disc-negated* (λ*x*::*nat*. *x* = *0*) *False* (*MatchAnd* (*Match 0*) (*MatchNot* (*Match 0*))) **by** *eval*
lemma *has-disc-negated* (λ*x*::*nat*. *x* = *0*) *True* (*MatchAnd* (*Match 0*) (*MatchNot* (*Match 1*))) **by** *eval*
lemma ¬ *has-disc-negated* (λ*x*::*nat*. *x* = *0*) *True* (*MatchAnd* (*Match 1*) (*MatchNot* (*Match 0*))) **by** *eval*
lemma *has-disc-negated* (λ*x*::*nat*. *x* = *0*) *True* (*MatchAnd* (*Match 0*) (*MatchNot* (*Match 0*))) **by** *eval*

— We want false on the right hand side, because this is how the algorithm should be started
lemma *has-disc-negated-MatchNot*:
has-disc-negated disc True (*MatchNot m*) ⟷ *has-disc-negated disc False m*
has-disc-negated disc True m ⟷ *has-disc-negated disc False* (*MatchNot m*)
by(*induction m*) (*simp-all*)

lemma *has-disc-negated-has-disc*: *has-disc-negated disc neg m* ⟹ *has-disc disc m*
apply(*induction m arbitrary*: *neg*)
apply(*simp-all split*: *if-split-asm*)
by *blast*

lemma *has-disc-negated-positiv-has-disc*: *has-disc-negated disc neg m* ∨ *has-disc-negated disc* (¬ *neg*) *m* ⟷ *has-disc disc m*
by(*induction disc neg m arbitrary*: *neg rule*:*has-disc-negated.induct*) *auto*

lemma *has-disc-negated-disj-split*:
has-disc-negated (λ*a*. *P a* ∨ *Q a*) *neg m* ⟷ *has-disc-negated P neg m* ∨ *has-disc-negated Q neg m*
apply(*induction* (λ*a*. *P a* ∨ *Q a*) *neg m rule*: *has-disc-negated.induct*)
apply(*simp-all*)
by *blast*

lemma *has-disc-alist-and*: *has-disc disc* (*alist-and as*) ⟷ (∃ *a* ∈ *set as*. *has-disc disc* (*negation-type-to-match-expr a*))
proof(*induction as rule*: *alist-and.induct*)
qed(*simp-all add*: *negation-type-to-match-expr-simps*)
lemma *has-disc-negated-alist-and*: *has-disc-negated disc neg* (*alist-and as*) ⟷ (∃ *a* ∈ *set as*. *has-disc-negated disc neg* (*negation-type-to-match-expr a*))
proof(*induction as rule*: *alist-and.induct*)
qed(*simp-all add*: *negation-type-to-match-expr-simps*)

lemma *has-disc-alist-and′*: *has-disc disc* (*alist-and′ as*) ⟷ (∃ *a* ∈ *set as*. *has-disc disc* (*negation-type-to-match-expr a*))
proof(*induction as rule*: *alist-and′.induct*)
qed(*simp-all add*: *negation-type-to-match-expr-simps*)
lemma *has-disc-negated-alist-and′*: *has-disc-negated disc neg* (*alist-and′ as*) ⟷

(∃ *a* ∈ *set as*. *has-disc-negated disc neg* (*negation-type-to-match-expr a*))
proof(*induction as rule*: *alist-and′.induct*)
qed(*simp-all add*: *negation-type-to-match-expr-simps*)

lemma *has-disc-alist-and′-append*:
has-disc disc′ (*alist-and′* (*ls1* @ *ls2*)) ⟷
has-disc disc′ (*alist-and′ ls1*) ∨ *has-disc disc′* (*alist-and′ ls2*)
apply(*induction ls1 arbitrary*: *ls2 rule*: *alist-and′.induct*)
apply(*simp-all*)
apply(*case-tac* [!] *ls2*)
apply(*simp-all*)
done
lemma *has-disc-negated-alist-and′-append*:
has-disc-negated disc′ neg (*alist-and′* (*ls1* @ *ls2*)) ⟷
has-disc-negated disc′ neg (*alist-and′ ls1*) ∨ *has-disc-negated disc′ neg* (*alist-and′ ls2*)
apply(*induction ls1 arbitrary*: *ls2 rule*: *alist-and′.induct*)
apply(*simp-all*)
apply(*case-tac* [!] *ls2*)
apply(*simp-all*)
done

lemma *match-list-to-match-expr-not-has-disc*:
∀ *a*. ¬ *disc* (*X a*) ⟹ ¬ *has-disc disc* (*match-list-to-match-expr* (*map* (*Match* ∘ *X*) *ls*))
apply(*induction ls*)
apply(*simp*; *fail*)
by(*simp add*: *MatchOr-def*)

lemma *matches* ((λ*x* -. *bool-to-ternary* (*disc x*)), (λ- -. *False*)) (*Match x*) *a p* ⟷ *has-disc disc* (*Match x*)
by(*simp add*: *match-raw-ternary bool-to-ternary-simps split*: *ternaryvalue.split*)

fun *normalized-n-primitive* :: ((′*a* ⇒ *bool*) × (′*a* ⇒ ′*b*)) ⇒ (′*b* ⇒ *bool*) ⇒ ′*a match-expr* ⇒ *bool* **where**
normalized-n-primitive - - *MatchAny* = *True* |
normalized-n-primitive (*disc*, *sel*) *n* (*Match P*) = (*if disc P then n* (*sel P*) *else True*) |
normalized-n-primitive (*disc*, *sel*) *n* (*MatchNot* (*Match P*)) = (*if disc P then False else True*) |
normalized-n-primitive (*disc*, *sel*) *n* (*MatchAnd m1 m2*) = (*normalized-n-primitive* (*disc*, *sel*) *n m1* ∧ *normalized-n-primitive* (*disc*, *sel*) *n m2*) |
normalized-n-primitive - - (*MatchNot* (*MatchAnd* - -)) = *False* |

normalized-n-primitive - - (*MatchNot* (*MatchNot* -)) = *False* |
normalized-n-primitive - - (*MatchNot MatchAny*) = *True*

lemma *normalized-nnf-match-opt-MatchAny-match-expr*:
normalized-nnf-match m ⟹ normalized-nnf-match (opt-MatchAny-match-expr m)
proof −
have *normalized-nnf-match m ⟹ normalized-nnf-match (opt-MatchAny-match-expr-once m)*
for *m* :: *'a match-expr*
by(*induction m rule*: *opt-MatchAny-match-expr-once.induct*) (*simp-all*)
thus *normalized-nnf-match m ⟹ normalized-nnf-match (opt-MatchAny-match-expr m)*
apply(*simp add*: *opt-MatchAny-match-expr-def*)
apply(*induction rule*: *repeat-stabilize-induct*)
by(*simp*)+
qed

lemma *normalized-n-primitive-opt-MatchAny-match-expr*:
normalized-n-primitive disc-sel f m ⟹ normalized-n-primitive disc-sel f (opt-MatchAny-match-expr m)
proof −

have *normalized-n-primitive disc-sel f m ⟹ normalized-n-primitive disc-sel f (opt-MatchAny-match-expr-once m)*
for *m*
proof −
{ **fix** *disc*::(*'a ⇒ bool*) **and** *sel*::(*'a ⇒ 'b*) **and** *n m1 m2*
have *normalized-n-primitive* (*disc*, *sel*) *n* (*opt-MatchAny-match-expr-once m1*) ⟹
normalized-n-primitive (*disc*, *sel*) *n* (*opt-MatchAny-match-expr-once m2*) ⟹
normalized-n-primitive (*disc*, *sel*) *n m1* ∧ *normalized-n-primitive* (*disc*, *sel*) *n m2* ⟹
normalized-n-primitive (*disc*, *sel*) *n* (*opt-MatchAny-match-expr-once* (*MatchAnd m1 m2*))
by(*induction* (*MatchAnd m1 m2*) *rule*: *opt-MatchAny-match-expr-once.induct*) (*auto*)
}**note** *x*=*this*
assume *normalized-n-primitive disc-sel f m*
thus *?thesis*
apply(*induction disc-sel f m rule*: *normalized-n-primitive.induct*)
apply *simp-all*
using *x* **by** *simp*
qed
from *this* **show**
normalized-n-primitive disc-sel f m ⟹ normalized-n-primitive disc-sel f (opt-MatchAny-match-expr m)
apply(*simp add*: *opt-MatchAny-match-expr-def*)
apply(*induction rule*: *repeat-stabilize-induct*)

by(*simp*)+
qed

lemma *normalized-n-primitive-imp-not-disc-negated*:
wf-disc-sel (*disc*,*sel*) *C* ⟹ *normalized-n-primitive* (*disc*,*sel*) *f m* ⟹ ¬ *has-disc-negated disc False m*
apply(*induction* (*disc*,*sel*) *f m rule*: *normalized-n-primitive.induct*)
by(*simp add*: *wf-disc-sel.simps split*: *if-split-asm*)+

lemma *normalized-n-primitive-alist-and*: *normalized-n-primitive disc-sel P* (*alist-and as*) ⟷
(∀ *a* ∈ *set as*. *normalized-n-primitive disc-sel P* (*negation-type-to-match-expr a*))
proof(*induction as*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a as*) **thus** *?case*
apply(*cases disc-sel*, *cases a*)
by(*simp-all add*: *negation-type-to-match-expr-simps*)
qed

lemma *normalized-n-primitive-alist-and'*: *normalized-n-primitive disc-sel P* (*alist-and' as*) ⟷
(∀ *a* ∈ *set as*. *normalized-n-primitive disc-sel P* (*negation-type-to-match-expr a*))
apply(*cases disc-sel*)
apply(*induction as rule*: *alist-and'.induct*)
by(*simp-all add*: *negation-type-to-match-expr-simps*)

lemma *not-has-disc-NegPos-map*: ∀ *a*. ¬ *disc* (*C a*) ⟹ ∀ *a*∈*set* (*NegPos-map C ls*).
¬ *has-disc disc* (*negation-type-to-match-expr a*)
by(*induction C ls rule*: *NegPos-map.induct*) (*simp add*: *negation-type-to-match-expr-def*)+

lemma *not-has-disc-negated-NegPos-map*: ∀ *a*. ¬ *disc* (*C a*) ⟹ ∀ *a*∈*set* (*NegPos-map C ls*).
¬ *has-disc-negated disc False* (*negation-type-to-match-expr a*)
by(*induction C ls rule*: *NegPos-map.induct*) (*simp add*: *negation-type-to-match-expr-def*)+

lemma *normalized-n-primitive-impossible-map*: ∀ *a*. ¬ *disc* (*C a*) ⟹
∀ *m*∈*set* (*map* (*Match* ∘ (*C* ∘ *x*)) *ls*).
normalized-n-primitive (*disc*, *sel*) *f m*
apply(*intro ballI*)
apply(*induction ls*)
apply(*simp*; *fail*)
apply(*simp*)
apply(*case-tac m*, *simp-all*)
apply(*fastforce*)

by *force*

lemma *normalized-n-primitive-alist-and'-append*:
normalized-n-primitive (*disc*, *sel*) *f* (*alist-and'* (*ls1* @ *ls2*)) ⟷
normalized-n-primitive (*disc*, *sel*) *f* (*alist-and'* *ls1*) ∧ *normalized-n-primitive* (*disc*, *sel*) *f* (*alist-and'* *ls2*)
apply(*induction ls1 arbitrary*: *ls2 rule*: *alist-and'.induct*)
apply(*simp-all*)
apply(*case-tac* [!] *ls2*)
apply(*simp-all*)
done

lemma *normalized-n-primitive-if-no-primitive*: *normalized-nnf-match m* ⟹ ¬ *has-disc disc m* ⟹
normalized-n-primitive (*disc*, *sel*) *f m*
by(*induction* (*disc*, *sel*) *f m rule*: *normalized-n-primitive.induct*) (*simp*)+

lemma *normalized-n-primitive-false-eq-notdisc*: *normalized-nnf-match m* ⟹
normalized-n-primitive (*disc*, *sel*) (λ-. *False*) *m* ⟷ ¬ *has-disc disc m*
proof −
have *normalized-nnf-match m* ⟹ *false* = (λ-. *False*) ⟹
¬ *has-disc disc m* ⟷ *normalized-n-primitive* (*disc*, *sel*) *false m* **for** *false*
by(*induction* (*disc*, *sel*) *false m rule*: *normalized-n-primitive.induct*)
(*simp*)+
thus *normalized-nnf-match m* ⟹ *?thesis* **by** *simp*
qed

lemma *normalized-n-primitive-MatchAnd-combine-map*: *normalized-n-primitive disc-sel f rst* ⟹
∀ *m'* ∈ (λ*spt*. *Match* (*C spt*)) ‘ *set pts*. *normalized-n-primitive disc-sel f m'* ⟹
m' ∈ (λ*spt*. *MatchAnd* (*Match* (*C spt*)) *rst*) ‘ *set pts* ⟹ *normalized-n-primitive disc-sel f m'*
by(*induction disc-sel f m' rule*: *normalized-n-primitive.induct*)
fastforce+

24.2 Primitive Extractor

The following function takes a tuple of functions ((*'a* ⇒ *bool*) × (*'a* ⇒ *'b*)) and a *'a match-expr*. The passed function tuple must be the discriminator and selector of the datatype package. *primitive-extractor* filters the *'a match-expr* and returns a tuple. The first element of the returned tuple is the filtered primitive matches, the second element is the remaining match expression.

It requires a *normalized-nnf-match*.

fun *primitive-extractor* :: ((*'a* ⇒ *bool*) × (*'a* ⇒ *'b*)) ⇒ *'a match-expr* ⇒ (*'b negation-type list* × *'a match-expr*) **where**
primitive-extractor - *MatchAny* = ([], *MatchAny*) |

```
primitive-extractor (disc,sel) (Match a) = (if disc a then ([Pos (sel a)], MatchAny)
else ([], Match a)) |
primitive-extractor (disc,sel) (MatchNot (Match a)) = (if disc a then ([Neg (sel
a)], MatchAny) else ([], MatchNot (Match a))) |
primitive-extractor C (MatchAnd ms1 ms2) = (
      let (a1', ms1') = primitive-extractor C ms1;
          (a2', ms2') = primitive-extractor C ms2
      in (a1'@a2', MatchAnd ms1' ms2')) |
primitive-extractor - - = undefined
```

The first part returned by *primitive-extractor*, here *as*: A list of primitive match expressions. For example, let m = *MatchAnd* (*Src ip1*) (*Dst ip2*) then, using the src (*disc*, *sel*), the result is [*ip1*]. Note that *Src* is stripped from the result.

The second part, here *ms* is the match expression which was not extracted. Together, the first and second part match iff m matches.

```
lemma primitive-extractor-fst-simp2:
  fixes m'::'a match-expr ⇒ 'a match-expr ⇒ 'a match-expr
  shows fst (case primitive-extractor (disc, sel) m1 of (a1', ms1') ⇒ case primi-
tive-extractor (disc, sel) m2 of (a2', ms2') ⇒ (a1' @ a2', m' ms1' ms2')) =
          fst (primitive-extractor (disc, sel) m1) @ fst (primitive-extractor (disc,
sel) m2)
     apply(cases primitive-extractor (disc, sel) m1, simp)
     apply(cases primitive-extractor (disc, sel) m2, simp)
     done

theorem primitive-extractor-correct: assumes
   normalized-nnf-match m and wf-disc-sel (disc, sel) C and primitive-extractor
(disc, sel) m = (as, ms)
  shows matches γ (alist-and (NegPos-map C as)) a p ∧ matches γ ms a p ⟷
matches γ m a p
  and normalized-nnf-match ms
  and ¬ has-disc disc ms
  and ∀ disc2. ¬ has-disc disc2 m ⟶ ¬ has-disc disc2 ms
 and ∀ disc2 sel2. normalized-n-primitive (disc2, sel2) P m ⟶ normalized-n-primitive
(disc2, sel2) P ms
  and ∀ disc2. ¬ has-disc-negated disc2 neg m ⟶ ¬ has-disc-negated disc2 neg ms
  and ¬ has-disc disc m ⟷ as = [] ∧ ms = m
  and ¬ has-disc-negated disc False m ⟷ getNeg as = []
  and has-disc disc m ⟹ as ≠ []
proof −
  — better simplification rule
  from assms have assm3': (as, ms) = primitive-extractor (disc, sel) m by simp
  with assms(1) assms(2) show matches γ (alist-and (NegPos-map C as)) a p ∧
matches γ ms a p ⟷ matches γ m a p
   proof(induction (disc, sel) m  arbitrary: as ms rule: primitive-extractor.induct)
    case 4 thus ?case
      apply(simp split: if-split-asm prod.split-asm add: NegPos-map-append)
```

apply(*auto simp add*: *alist-and-append bunch-of-lemmata-about-matches*)
done
qed(*simp-all add*: *bunch-of-lemmata-about-matches wf-disc-sel.simps split*: *if-split-asm*)

from *assms*(*1*) *assm3'* **show** *normalized-nnf-match ms*
proof(*induction* (*disc*, *sel*) *m arbitrary*: *as ms rule*: *primitive-extractor.induct*)
case *2* **thus** *?case* **by**(*simp split*: *if-split-asm*)
next
case *3* **thus** *?case* **by**(*simp split*: *if-split-asm*)
next
case *4* **thus** *?case*
apply(*clarify*)
apply(*simp split*: *prod.split-asm*)
done
qed(*simp-all*)

from *assms*(*1*) *assm3'* **show** ¬ *has-disc disc ms*
proof(*induction* (*disc*, *sel*) *m arbitrary*: *as ms rule*: *primitive-extractor.induct*)
qed(*simp-all split*: *if-split-asm prod.split-asm*)

from *assms*(*1*) *assm3'* **show** ∀ *disc2*. ¬ *has-disc disc2 m* ⟶ ¬ *has-disc disc2 ms*
proof(*induction* (*disc*, *sel*) *m arbitrary*: *as ms rule*: *primitive-extractor.induct*)
case *2* **thus** *?case* **by**(*simp split*: *if-split-asm*)
next
case *3* **thus** *?case* **by**(*simp split*: *if-split-asm*)
next
case *4* **thus** *?case* **by**(*simp split*: *prod.split-asm*)
qed(*simp-all*)

from *assms*(*1*) *assm3'* **show** ∀ *disc2*. ¬ *has-disc-negated disc2 neg m* ⟶ ¬ *has-disc-negated disc2 neg ms*
proof(*induction* (*disc*, *sel*) *m arbitrary*: *as ms rule*: *primitive-extractor.induct*)
case *2* **thus** *?case* **by**(*simp split*: *if-split-asm*)
next
case *3* **thus** *?case* **by**(*simp split*: *if-split-asm*)
next
case *4* **thus** *?case* **by**(*simp split*: *prod.split-asm*)
qed(*simp-all*)

from *assms*(*1*) *assm3'* **show** ∀ *disc2 sel2*. *normalized-n-primitive* (*disc2*, *sel2*) *P m* ⟶ *normalized-n-primitive* (*disc2*, *sel2*) *P ms*
apply(*induction* (*disc*, *sel*) *m arbitrary*: *as ms rule*: *primitive-extractor.induct*)
apply(*simp*)
apply(*simp split*: *if-split-asm*)
apply(*simp split*: *if-split-asm*)

```
      apply(simp split: prod.split-asm)
     apply(simp-all)
    done

   from assms(1) assm3' show ¬ has-disc disc m ⟷ as = [] ∧ ms = m
   proof(induction (disc, sel) m  arbitrary: as ms rule: primitive-extractor.induct)
    case 2 thus ?case by(simp split: if-split-asm)
    next
    case 3 thus ?case by(simp split: if-split-asm)
    next
    case 4 thus ?case by(auto split: prod.split-asm)
    qed(simp-all)

   from assms(1) assm3' show ¬ has-disc-negated disc False m ⟷ getNeg as =
[]
   proof(induction (disc, sel) m  arbitrary: as ms rule: primitive-extractor.induct)
    case 2 thus ?case by(simp split: if-split-asm)
    next
    case 3 thus ?case by(simp split: if-split-asm)
    next
    case 4 thus ?case by(simp add: getNeg-append split: prod.split-asm)
    qed(simp-all)

   from assms(1) assm3' show has-disc disc m ⟹ as ≠ []
   proof(induction (disc, sel) m  arbitrary: as ms rule: primitive-extractor.induct)
    case 4 thus ?case apply(simp split: prod.split-asm)
     by metis
    qed(simp-all)
qed


lemma has-disc-negated-primitive-extractor:
  assumes normalized-nnf-match m
 shows has-disc-negated disc False m ⟷ (∃ a. Neg a ∈ set (fst (primitive-extractor
(disc, sel) m)))
proof −
  obtain as ms where asms: primitive-extractor (disc, sel) m = (as, ms) by
fastforce
  hence has-disc-negated disc False m ⟷ (∃ a. Neg a ∈ set as)
    using assms proof(induction m arbitrary: as ms)
    case Match thus ?case
      by(simp split: if-split-asm) fastforce
    next
    case (MatchNot m)
     thus ?case
     proof(induction m)
     case Match thus ?case by (simp, fastforce)
     qed(simp-all)
    next
```

```
    case (MatchAnd m1 m2) thus ?case
      apply(cases primitive-extractor (disc, sel) m1)
      apply(cases primitive-extractor (disc, sel) m2)
      by auto
  qed(simp-all split: if-split-asm)
  thus ?thesis using asms by simp
qed




lemma primitive-extractor-reassemble-preserves:
  wf-disc-sel (disc, sel) C ⟹
   normalized-nnf-match m ⟹
   P m ⟹
   P MatchAny ⟹
  primitive-extractor (disc, sel) m = (as, ms) ⟹ — turn eqality around to simplify
proof
   (⋀m1 m2. P (MatchAnd m1 m2) ⟷ P m1 ∧ P m2) ⟹
   (⋀ls1 ls2. P (alist-and' (ls1 @ ls2)) ⟷ P (alist-and' ls1) ∧ P (alist-and' ls2))
⟹
   P (alist-and' (NegPos-map C as))
  proof(induction (disc, sel) m  arbitrary: as ms rule: primitive-extractor.induct)
  case 2 thus ?case
    apply(simp split: if-split-asm)
    apply(clarify)
    by(simp add: wf-disc-sel.simps)
  next
  case 3 thus ?case
    apply(simp split: if-split-asm)
    apply(clarify)
    by(simp add: wf-disc-sel.simps)
  next
  case (4 m1 m2 as ms)
    from 4 show ?case
      apply(simp)
      apply(simp split: prod.split-asm)
      apply(clarify)
      apply(simp add: NegPos-map-append)
    done
qed(simp-all split: if-split-asm)

lemma primitive-extractor-reassemble-not-has-disc:
  wf-disc-sel (disc, sel) C ⟹
   normalized-nnf-match m ⟹ ¬ has-disc disc' m ⟹
   primitive-extractor (disc, sel) m = (as, ms) ⟹
    ¬ has-disc disc' (alist-and' (NegPos-map C as))
  apply(rule primitive-extractor-reassemble-preserves)
        by(simp-all add: NegPos-map-append has-disc-alist-and'-append)
```

lemma *primitive-extractor-reassemble-not-has-disc-negated*:
wf-disc-sel (*disc*, *sel*) *C* $\Longrightarrow$
normalized-nnf-match m $\Longrightarrow$ $\neg$ *has-disc-negated disc′ neg m* $\Longrightarrow$
primitive-extractor (*disc*, *sel*) *m* = (*as*, *ms*) $\Longrightarrow$
$\neg$ *has-disc-negated disc′ neg* (*alist-and′* (*NegPos-map C as*))
apply(*rule primitive-extractor-reassemble-preserves*)
by(*simp-all add*: *NegPos-map-append has-disc-negated-alist-and′-append*)

lemma *primitive-extractor-reassemble-normalized-n-primitive*:
wf-disc-sel (*disc*, *sel*) *C* $\Longrightarrow$
normalized-nnf-match m $\Longrightarrow$ *normalized-n-primitive* (*disc1*, *sel1*) *f m* $\Longrightarrow$
primitive-extractor (*disc*, *sel*) *m* = (*as*, *ms*) $\Longrightarrow$
normalized-n-primitive (*disc1*, *sel1*) *f* (*alist-and′* (*NegPos-map C as*))
apply(*rule primitive-extractor-reassemble-preserves*)
by(*simp-all add*: *NegPos-map-append normalized-n-primitive-alist-and′-append*)

lemma *primitive-extractor-matchesE*: *wf-disc-sel* (*disc*,*sel*) *C* $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *primitive-extractor* (*disc*, *sel*) *m* = (*as*, *ms*)
$\Longrightarrow$
(*normalized-nnf-match ms* $\Longrightarrow$ $\neg$ *has-disc disc ms* $\Longrightarrow$ ($\forall$ *disc2*. $\neg$ *has-disc disc2 m* $\longrightarrow$ $\neg$ *has-disc disc2 ms*) $\Longrightarrow$ *matches-other* $\longleftrightarrow$ *matches* γ *ms a p*)
$\Longrightarrow$
matches γ (*alist-and* (*NegPos-map C as*)) *a p* $\wedge$ *matches-other* $\longleftrightarrow$ *matches* γ *m a p*
using *primitive-extractor-correct*(*1*,*2*,*3*,*4*) **by** *metis*

lemma *primitive-extractor-matches-lastE*: *wf-disc-sel* (*disc*,*sel*) *C* $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *primitive-extractor* (*disc*, *sel*) *m* = (*as*, *ms*)
$\Longrightarrow$
(*normalized-nnf-match ms* $\Longrightarrow$ $\neg$ *has-disc disc ms* $\Longrightarrow$ ($\forall$ *disc2*. $\neg$ *has-disc disc2 m* $\longrightarrow$ $\neg$ *has-disc disc2 ms*) $\Longrightarrow$ *matches* γ *ms a p*)
$\Longrightarrow$
matches γ (*alist-and* (*NegPos-map C as*)) *a p* $\longleftrightarrow$ *matches* γ *m a p*
using *primitive-extractor-correct*(*1*,*2*,*3*,*4*) **by** *metis*

The lemmas $[\![$*wf-disc-sel* (*?disc*, *?sel*) *?C*; *normalized-nnf-match ?m*; *primitive-extractor* (*?disc*, *?sel*) *?m* = (*?as*, *?ms*); $[\![$*normalized-nnf-match ?ms*; $\neg$ *has-disc ?disc ?ms*; $\forall$ *disc2*. $\neg$ *has-disc disc2 ?m* $\longrightarrow$ $\neg$ *has-disc disc2 ?ms*$]\!]$ $\Longrightarrow$ *?matches-other* = *matches ?*γ *?ms ?a ?p*$]\!]$ $\Longrightarrow$ (*matches ?*γ (*alist-and* (*NegPos-map ?C ?as*)) *?a ?p* $\wedge$ *?matches-other*) = *matches ?*γ *?m ?a ?p* and $[\![$*wf-disc-sel* (*?disc*, *?sel*) *?C*; *normalized-nnf-match ?m*; *primitive-extractor* (*?disc*, *?sel*) *?m* = (*?as*, *?ms*); $[\![$*normalized-nnf-match ?ms*; $\neg$ *has-disc ?disc ?ms*; $\forall$ *disc2*. $\neg$ *has-disc disc2 ?m* $\longrightarrow$ $\neg$ *has-disc disc2 ?ms*$]\!]$ $\Longrightarrow$ *matches ?*γ *?ms ?a ?p*$]\!]$ $\Longrightarrow$ *matches ?*γ (*alist-and* (*NegPos-map ?C ?as*)) *?a ?p* = *matches ?*γ *?m ?a ?p* can be used as erule to solve goals about con-

secutive application of *primitive-extractor*. They should be used as *primitive-extractor-matchesE*[*OF wf-disc-sel-for-first-extracted-thing*].

24.3 Normalizing and Optimizing Primitives

Normalize primitives by a function f with type *'b negation-type list* ⇒ *'b list*. *'b* is a primitive type, e.g. ipt-ipv4range. f takes a conjunction list of negated primitives and must compress them such that:

1. no negation occurs in the output
2. the output is a disjunction of the primitives, i.e. multiple primitives in one rule are compressed to at most one primitive (leading to multiple rules)

Example with IP addresses:

```
f [10.8.0.0/16, 10.0.0.0/8] = [10.0.0.0/8]  f compresses to one range
f [10.0.0.0, 192.168.0.01] = []    range is empty, rule can be dropped
f [Neg 41] = [{0..40}, {42..ipv4max}]   one rule is translated into multiple r
f [Neg 41, {20..50}, {30..50}] = [{30..40}, {42..50}]   input: conjunction lis
```

definition *normalize-primitive-extract* :: ((*'a* ⇒ *bool*) × (*'a* ⇒ *'b*)) ⇒
(*'b* ⇒ *'a*) ⇒
(*'b negation-type list* ⇒ *'b list*) ⇒
'a match-expr ⇒
'a match-expr list **where**
normalize-primitive-extract (*disc-sel*) *C f m* ≡ (*case primitive-extractor* (*disc-sel*) *m*
of (*spts*, *rst*) ⇒ *map* (λ*spt*. (*MatchAnd* (*Match* (*C spt*))) *rst*) (*f spts*))

If f has the properties described above, then *normalize-primitive-extract* is a valid transformation of a match expression

lemma *normalize-primitive-extract*: **assumes** *normalized-nnf-match m* **and** *wf-disc-sel disc-sel C* **and**
∀ *ml*. (*match-list* γ (*map* (*Match* ∘ *C*) (*f ml*)) *a p* ⟷ *matches* γ (*alist-and* (*NegPos-map C ml*)) *a p*)
shows *match-list* γ (*normalize-primitive-extract disc-sel C f m*) *a p* ⟷ *matches* γ *m a p*
proof −
obtain *as ms* **where** *pe*: *primitive-extractor disc-sel m* = (*as*, *ms*) **by** *fastforce*

from *pe primitive-extractor-correct*(*1*)[*OF assms*(*1*), **where** γ=γ **and** *a*=*a* **and** *p*=*p*] *assms*(*2*) **have**
matches γ *m a p* ⟷ *matches* γ (*alist-and* (*NegPos-map C as*)) *a p* ∧ *matches* γ *ms a p* **by**(*cases disc-sel*, *blast*)

also have ... $\longleftrightarrow$ *match-list* γ (*map* (*Match* $\circ$ *C*) (*f as*)) *a p* $\wedge$ *matches* γ *ms a p* **using** *assms*(*3*) **by** *simp*
also have ... $\longleftrightarrow$ *match-list* γ (*map* (λ*spt*. *MatchAnd* (*Match* (*C spt*)) *ms*) (*f as*)) *a p*
by(*simp add*: *match-list-matches bunch-of-lemmata-about-matches*)
also have ... $\longleftrightarrow$ *match-list* γ (*normalize-primitive-extract disc-sel C f m*) *a p*
by(*simp add*: *normalize-primitive-extract-def pe*)
finally show *?thesis* **by** *simp*
qed

thm *match-list-semantics*[*of* γ (*map* (*Match* $\circ$ *C*) (*f ml*)) *a p* [(*alist-and* (*NegPos-map C ml*))]]

corollary *normalize-primitive-extract-semantics*: **assumes** *normalized-nnf-match m* **and** *wf-disc-sel disc-sel C* **and**
$\forall$ *ml*. (*match-list* γ (*map* (*Match* $\circ$ *C*) (*f ml*)) *a p* $\longleftrightarrow$ *matches* γ (*alist-and* (*NegPos-map C ml*)) *a p*)
shows *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*normalize-primitive-extract disc-sel C f m*)) *s* =
approximating-bigstep-fun γ *p* [*Rule m a*] *s*
proof −
from *normalize-primitive-extract*[*OF assms*(*1*) *assms*(*2*) *assms*(*3*)] **have**
match-list γ (*normalize-primitive-extract disc-sel C f m*) *a p* = *matches* γ *m a p* **.**
also have ... $\longleftrightarrow$ *match-list* γ [*m*] *a p* **by** *simp*
finally show *?thesis* **using** *match-list-semantics*[*of* γ (*normalize-primitive-extract disc-sel C f m*) *a p* [*m*]] **by** *simp*
qed

lemma *normalize-primitive-extract-preserves-nnf-normalized*:
assumes *normalized-nnf-match m*
and *wf-disc-sel* (*disc*, *sel*) *C*
shows $\forall$ *mn* $\in$ *set* (*normalize-primitive-extract* (*disc*, *sel*) *C f m*). *normalized-nnf-match mn*
proof
fix *mn*
assume *assm2*: *mn* $\in$ *set* (*normalize-primitive-extract* (*disc*, *sel*) *C f m*)
obtain *as ms* **where** *as-ms*: *primitive-extractor* (*disc*, *sel*) *m* = (*as*, *ms*) **by** *fastforce*
from *primitive-extractor-correct*(*2*)[*OF assms*(*1*) *assms*(*2*) *as-ms*] **have** *normalized-nnf-match ms* **by** *simp*
from *assm2 as-ms* **have** *normalize-primitive-extract-unfolded*: *mn* $\in$ ((λ*spt*. *MatchAnd* (*Match* (*C spt*)) *ms*) ‘ *set* (*f as*))
unfolding *normalize-primitive-extract-def* **by** *force*
with ‹*normalized-nnf-match ms*› **show** *normalized-nnf-match mn* **by** *fastforce*
qed

lemma *normalize-rules-primitive-extract-preserves-nnf-normalized*:
∀ *r* ∈ *set rs*. *normalized-nnf-match* (*get-match r*) ⟹ *wf-disc-sel disc-sel C* ⟹
∀ *r* ∈ *set* (*normalize-rules* (*normalize-primitive-extract disc-sel C f*) *rs*). *normalized-nnf-match* (*get-match r*)
apply(*rule normalize-rules-preserves*[**where** *P*=*normalized-nnf-match* **and** *f*=(*normalize-primitive-extract disc-sel C f*)])
apply(*simp*; *fail*)
apply(*cases disc-sel*)
using *normalize-primitive-extract-preserves-nnf-normalized* **by** *fast*

If something is normalized for disc2 and disc2 ≠ disc1 and we do something on disc1, then disc2 remains normalized

lemma *normalize-primitive-extract-preserves-unrelated-normalized-n-primitive*:
assumes *normalized-nnf-match m*
and *normalized-n-primitive* (*disc2*, *sel2*) *P m*
and *wf-disc-sel* (*disc1*, *sel1*) *C*
and ∀ *a*. ¬ *disc2* (*C a*) — disc1 and disc2 match for different stuff. e.g. *Src-Ports* and *Dst-Ports*
shows ∀ *mn* ∈ *set* (*normalize-primitive-extract* (*disc1*, *sel1*) *C f m*). *normalized-n-primitive* (*disc2*, *sel2*) *P mn*
proof
fix *mn*
assume *assm2*: *mn* ∈ *set* (*normalize-primitive-extract* (*disc1*, *sel1*) *C f m*)
obtain *as ms* **where** *as-ms*: *primitive-extractor* (*disc1*, *sel1*) *m* = (*as*, *ms*) **by** *fastforce*
from *as-ms primitive-extractor-correct*[*OF assms*(*1*) *assms*(*3*)] **have**
¬ *has-disc disc1 ms*
and *normalized-n-primitive* (*disc2*, *sel2*) *P ms*
apply −
apply(*fast*)
using *assms*(*2*) **by**(*fast*)
from *assm2 as-ms* **have** *normalize-primitive-extract-unfolded*: *mn* ∈ ((λ*spt*. *MatchAnd* (*Match* (*C spt*)) *ms*) ‘ *set* (*f as*))
unfolding *normalize-primitive-extract-def* **by** *force*

from *normalize-primitive-extract-unfolded* **obtain** *Casms* **where** *Casms*: *mn* = (*MatchAnd* (*Match* (*C Casms*)) *ms*) **by** *blast*

from ‹*normalized-n-primitive* (*disc2*, *sel2*) *P ms*› *assms*(*4*) **have** *normalized-n-primitive* (*disc2*, *sel2*) *P* (*MatchAnd* (*Match* (*C Casms*)) *ms*)
by(*simp*)

with *Casms* **show** *normalized-n-primitive* (*disc2*, *sel2*) *P mn* **by** *blast*
qed

lemma *normalize-primitive-extract-normalizes-n-primitive*:
fixes *disc*::(′*a* ⇒ *bool*) **and** *sel*::(′*a* ⇒ ′*b*) **and** *f*::(′*b negation-type list* ⇒ ′*b list*)
assumes *normalized-nnf-match m*

and *wf-disc-sel* (*disc*, *sel*) *C*
and *np*: ∀ *as*. (∀ *a'* ∈ *set* (*f as*). *P a'*)
shows ∀ *m'* ∈ *set* (*normalize-primitive-extract* (*disc*, *sel*) *C f m*). *normalized-n-primitive* (*disc*, *sel*) *P m'*
proof
fix *m'* **assume** *a*: *m'*∈*set* (*normalize-primitive-extract* (*disc*, *sel*) *C f m*)

have *nnf*: ∀ *m'* ∈ *set* (*normalize-primitive-extract* (*disc*, *sel*) *C f m*). *normalized-nnf-match m'*
using *normalize-primitive-extract-preserves-nnf-normalized assms* **by** *blast*
with *a* **have** *normalized-m'*: *normalized-nnf-match m'* **by** *simp*

from *a* **obtain** *as ms* **where** *as-ms*: *primitive-extractor* (*disc*, *sel*) *m* = (*as*, *ms*)
unfolding *normalize-primitive-extract-def* **by** *fastforce*
with *a* **have** *prems*: *m'* ∈ *set* (*map* (λ*spt*. *MatchAnd* (*Match* (*C spt*)) *ms*) (*f as*))
unfolding *normalize-primitive-extract-def* **by** *simp*

from *primitive-extractor-correct*(*2*)[*OF assms*(*1*) *assms*(*2*) *as-ms*] **have** *normalized-nnf-match ms* **.**

show *normalized-n-primitive* (*disc*, *sel*) *P m'*
proof(*cases f as* = [])
case *True* **thus** *normalized-n-primitive* (*disc*, *sel*) *P m'* **using** *prems* **by** *simp*
next
case *False*
with *prems* **obtain** *spt* **where** *m'* = *MatchAnd* (*Match* (*C spt*)) *ms* **and** *spt* ∈ *set* (*f as*) **by** *auto*

from *primitive-extractor-correct*(*3*)[*OF assms*(*1*) *assms*(*2*) *as-ms*] **have** ¬ *has-disc disc ms* **.**
with ‹*normalized-nnf-match ms*› **have** *normalized-n-primitive* (*disc*, *sel*) *P ms*
by(*induction* (*disc*, *sel*) *P ms rule*: *normalized-n-primitive.induct*) *simp-all*

from ‹*wf-disc-sel* (*disc*, *sel*) *C*› **have** (*sel* (*C spt*)) = *spt* **by**(*simp add*: *wf-disc-sel.simps*)
with *np* ‹*spt* ∈ *set* (*f as*)› **have** *P* (*sel* (*C spt*)) **by** *simp*

show *normalized-n-primitive* (*disc*, *sel*) *P m'*
apply(*simp add*: ‹*m'* = *MatchAnd* (*Match* (*C spt*)) *ms*›)
apply(*rule conjI*)
apply(*simp-all add*: ‹*normalized-n-primitive* (*disc*, *sel*) *P ms*›)
apply(*simp add*: ‹*P* (*sel* (*C spt*))›)
done
qed
qed

lemma *primitive-extractor-negation-type-matching1*:
assumes *wf*: *wf-disc-sel* (*disc*, *sel*) *C*
and *normalized*: *normalized-nnf-match m*
and *a1*: *primitive-extractor* (*disc*, *sel*) *m* = (*as*, *rest*)
and *a2*: *matches* γ *m a p*
shows ($\forall$ *m*$\in$*set* (*map C* (*getPos as*)). *matches* γ (*Match m*) *a p*) $\wedge$
($\forall$ *m*$\in$*set* (*map C* (*getNeg as*)). *matches* γ (*MatchNot* (*Match m*)) *a p*)
proof −
from *primitive-extractor-correct*(*1*)[*OF normalized wf a1*] *a2* **have**
matches γ (*alist-and* (*NegPos-map C as*)) *a p* $\wedge$ *matches* γ *rest a p* **by** *fast*
hence *matches* γ (*alist-and* (*NegPos-map C as*)) *a p* **by** *blast*
with *Negation-Type-Matching.matches-alist-and* **have**
($\forall$ *m*$\in$*set* (*getPos* (*NegPos-map C as*)). *matches* γ (*Match m*) *a p*) $\wedge$
($\forall$ *m*$\in$*set* (*getNeg* (*NegPos-map C as*)). *matches* γ (*MatchNot* (*Match m*)) *a p*) **by** *metis*
with *getPos-NegPos-map-simp2 getNeg-NegPos-map-simp2* **show** *?thesis* **by** *metis*
qed

normalized-n-primitive does NOT imply *normalized-nnf-match*

lemma $\exists$ *m*. *normalized-n-primitive disc-sel f m* $\longrightarrow$ $\neg$ *normalized-nnf-match m*
by(*rule-tac x*=*MatchNot MatchAny* **in** *exI*) (*simp*)

lemma *remove-unknowns-generic-not-has-disc*: $\neg$ *has-disc C m* $\Longrightarrow$ $\neg$ *has-disc C* (*remove-unknowns-generic* γ *a m*)
by(*induction* γ *a m rule*: *remove-unknowns-generic.induct*) (*simp-all add*: *remove-unknowns-generic-simps2*)

lemma *remove-unknowns-generic-not-has-disc-negated*: $\neg$ *has-disc-negated C neg m* $\Longrightarrow$ $\neg$ *has-disc-negated C neg* (*remove-unknowns-generic* γ *a m*)
by(*induction* γ *a m rule*: *remove-unknowns-generic.induct*) (*simp-all add*: *remove-unknowns-generic-simps2*)

lemma *remove-unknowns-generic-normalized-n-primitive*: *normalized-n-primitive disc-sel f m* $\Longrightarrow$
normalized-n-primitive disc-sel f (*remove-unknowns-generic* γ *a m*)
proof(*induction* γ *a m rule*: *remove-unknowns-generic.induct*)
case *6* **thus** *?case* **by**(*case-tac disc-sel*, *simp add*: *remove-unknowns-generic-simps2*)
qed(*simp-all add*: *remove-unknowns-generic-simps2*)

lemma *normalize-match-preserves-disc-negated*:
shows ($\exists$ *m-DNF* $\in$ *set* (*normalize-match m*). *has-disc-negated disc neg m-DNF*) $\Longrightarrow$ *has-disc-negated disc neg m*
proof(*induction m rule*: *normalize-match.induct*)
case *3* **thus** *?case* **by** (*simp*) *blast*

next
case *4*
from *4* **show** *?case* **by**(*simp*) *blast*
qed(*simp-all*)

has-disc-negated is a structural property and *normalize-match* is a semantical property. *normalize-match* removes subexpressions which cannot match. Thus, we cannot show (without complicated assumptions) the opposite direction of ∃ *m-DNF*∈*set* (*normalize-match ?m*). *has-disc-negated ?disc ?neg m-DNF* ⟹ *has-disc-negated ?disc ?neg ?m*, because a negated primitive might occur in a subexpression which will be optimized away.

corollary *i-m-giving-this-a-funny-name-so-i-can-thank-my-future-me-when-sledgehammer-will-find-this-one-do*
¬ *has-disc-negated disc neg m* ⟹ ∀ *m-DNF* ∈ *set* (*normalize-match m*). ¬ *has-disc-negated disc neg m-DNF*
using *normalize-match-preserves-disc-negated* **by** *blast*

lemma *not-has-disc-opt-MatchAny-match-expr*:
¬ *has-disc disc m* ⟹ ¬ *has-disc disc* (*opt-MatchAny-match-expr m*)
proof −
have ¬ *has-disc disc m* ⟹ ¬ *has-disc disc* (*opt-MatchAny-match-expr-once m*) **for** *m*
by(*induction m rule*: *opt-MatchAny-match-expr-once.induct*) *simp-all*
thus ¬ *has-disc disc m* ⟹ ¬ *has-disc disc* (*opt-MatchAny-match-expr m*)
apply(*simp add*: *opt-MatchAny-match-expr-def*)
apply(*rule repeat-stabilize-induct*)
by(*simp*)+
qed
lemma *not-has-disc-negated-opt-MatchAny-match-expr*:
¬ *has-disc-negated disc neg m* ⟹ ¬ *has-disc-negated disc neg* (*opt-MatchAny-match-expr m*)
proof −
have ¬ *has-disc-negated disc neg m* ⟹ ¬ *has-disc-negated disc neg* (*opt-MatchAny-match-expr-once m*)
for *m*
by(*induction m arbitrary*: *neg rule*:*opt-MatchAny-match-expr-once.induct*) (*simp-all*)
thus ¬ *has-disc-negated disc neg m* ⟹ ¬ *has-disc-negated disc neg* (*opt-MatchAny-match-expr m*)
apply(*simp add*: *opt-MatchAny-match-expr-def*)
apply(*rule repeat-stabilize-induct*)
by(*simp*)+
qed

lemma *normalize-match-preserves-nodisc*:
¬ *has-disc disc m* ⟹ *m′* ∈ *set* (*normalize-match m*) ⟹ ¬ *has-disc disc m′*
proof −

have ¬ *has-disc disc m* ⟶ (∀ *m′* ∈ *set* (*normalize-match m*). ¬ *has-disc disc m′*)

by(*induction m rule*: *normalize-match.induct*) (*safe*,*auto*) — need safe, otherwise simplifier loops
thus ¬ *has-disc disc m* ⟹ *m'* ∈ *set* (*normalize-match m*) ⟹ ¬ *has-disc disc m'* **by** *blast*
qed

lemma *not-has-disc-normalize-match*:
¬ *has-disc-negated disc neg m* ⟹ *m'* ∈ *set* (*normalize-match m*) ⟹ ¬ *has-disc-negated disc neg m'*
using *i-m-giving-this-a-funny-name-so-i-can-thank-my-future-me-when-sledgehammer-will-find-this-one-day*
by *blast*

lemma *normalize-match-preserves-normalized-n-primitive*:
normalized-n-primitive disc-sel f rst ⟹
∀ *m* ∈ *set* (*normalize-match rst*). *normalized-n-primitive disc-sel f m*
apply(*cases disc-sel*, *simp*)
apply(*induction rst rule*: *normalize-match.induct*)
apply(*simp*; *fail*)
apply(*simp*; *fail*)
apply(*simp*; *fail*)
using *normalized-n-primitive.simps*(*5*) **apply** *metis*
by *simp*+

24.4 Optimizing a match expression

Optimizes a match expression with a function that takes *'b negation-type list* and returns (*'b list* × *'b list*) *option*. The function should return *None* if the match expression cannot match. It returns *Some* (*as-pos*, *as-neg*) where *as-pos* and *as-neg* are lists of primitives. Positive and Negated. The result is one match expression.

In contrast *normalize-primitive-extract* returns a list of match expression, to be read es their disjunction.

definition *compress-normalize-primitive* :: ((*'a* ⇒ *bool*) × (*'a* ⇒ *'b*)) ⇒ (*'b* ⇒ *'a*) ⇒
(*'b negation-type list* ⇒ (*'b list* × *'b list*) *option*) ⇒
'a match-expr ⇒ *'a match-expr option* **where**
compress-normalize-primitive disc-sel C f m ≡ (*case primitive-extractor disc-sel m of* (*as*, *rst*) ⇒
(*map-option* (λ(*as-pos*, *as-neg*). *MatchAnd*
(*alist-and'* (*NegPos-map C* ((*map Pos as-pos*)@(*map Neg as-neg*))))
rst
) (*f as*)))

lemma *compress-normalize-primitive-nnf*: *wf-disc-sel disc-sel C* ⟹

normalized-nnf-match m $\Longrightarrow$ *compress-normalize-primitive disc-sel C f m = Some m'* $\Longrightarrow$
normalized-nnf-match m'
apply(*case-tac primitive-extractor disc-sel m*)
apply(*simp add*: *compress-normalize-primitive-def*)
apply(*clarify*)
apply (*simp add*: *normalized-nnf-match-alist-and'*)
apply(*cases disc-sel*, *simp*)
using *primitive-extractor-correct*(*2*) **by** *blast*

lemma *compress-normalize-primitive-not-introduces-C*:
assumes *notdisc*: $\neg$ *has-disc disc m*
and *wf*: *wf-disc-sel* (*disc*,*sel*) *C'*
and *nm*: *normalized-nnf-match m*
and *some*: *compress-normalize-primitive* (*disc*,*sel*) *C f m = Some m'*
and *f-preserves*: $\bigwedge$*as-pos as-neg. f* [] = *Some* (*as-pos*, *as-neg*) $\Longrightarrow$ *as-pos* = [] $\wedge$ *as-neg* = []
shows $\neg$ *has-disc disc m'*
proof −
obtain *as ms* **where** *asms*: *primitive-extractor* (*disc*, *sel*) *m* = (*as*, *ms*) **by** *fastforce*
from *notdisc primitive-extractor-correct*(*4*)[*OF nm wf asms*] **have** *1*: $\neg$ *has-disc disc ms* **by** *simp*
from *notdisc primitive-extractor-correct*(*7*)[*OF nm wf asms*] **have** *2*: *as* = [] $\wedge$ *ms* = *m* **by** *simp*
from *1 2 some* **show** *?thesis* **by**(*auto dest*: *f-preserves simp add*: *compress-normalize-primitive-def asms*)
qed

lemma *compress-normalize-primitive-not-introduces-C-negated*:
assumes *notdisc*: $\neg$ *has-disc-negated disc False m*
and *wf*: *wf-disc-sel* (*disc*,*sel*) *C*
and *nm*: *normalized-nnf-match m*
and *some*: *compress-normalize-primitive* (*disc*,*sel*) *C f m = Some m'*
and *f-preserves*: $\bigwedge$*as as-pos as-neg. f as = Some* (*as-pos*, *as-neg*) $\Longrightarrow$ *getNeg as* = [] $\Longrightarrow$ *as-neg* = []
shows $\neg$ *has-disc-negated disc False m'*
proof −
obtain *as ms* **where** *asms*: *primitive-extractor* (*disc*,*sel*) *m* = (*as*, *ms*) **by** *fastforce*
from *notdisc primitive-extractor-correct*(*6*)[*OF nm wf asms*] **have** *1*: $\neg$ *has-disc-negated disc False ms* **by** *simp*
from *asms notdisc has-disc-negated-primitive-extractor*[*OF nm*, **where** *disc=disc* **and** *sel=sel*] **have**
$\forall$ *a. Neg a* $\notin$ *set as* **by**(*simp*)
hence *getNeg as* = [] **by** (*meson NegPos-set*(*5*) *image-subset-iff last-in-set*)
with *f-preserves* **have** *f-preserves'*: $\bigwedge$*as-pos as-neg. f as = Some* (*as-pos*, *as-neg*) $\Longrightarrow$ *as-neg* = [] **by** *simp*

from *1* **have** $\bigwedge$ *a b.*$\neg$ *has-disc-negated disc False* (*MatchAnd* (*alist-and'* (*NegPos-map C* (*map Pos a*))) *ms*)
by(*simp add*: *has-disc-negated-alist-and' NegPos-map-map-Pos negation-type-to-match-expr-simps*)
with *some* **show** *?thesis* **by**(*auto dest*: *f-preserves' simp add*: *compress-normalize-primitive-def asms*)
qed

lemma *compress-normalize-primitive-Some*:
assumes *normalized*: *normalized-nnf-match m*
and *wf*: *wf-disc-sel* (*disc*,*sel*) *C*
and *some*: *compress-normalize-primitive* (*disc*,*sel*) *C f m* = *Some m'*
and *f-correct*: $\bigwedge$*as as-pos as-neg. f as* = *Some* (*as-pos*, *as-neg*) $\Longrightarrow$
matches γ (*alist-and* (*NegPos-map C* ((*map Pos as-pos*)@(*map Neg as-neg*)))) *a p* $\longleftrightarrow$
matches γ (*alist-and* (*NegPos-map C as*)) *a p*
shows *matches* γ *m' a p* $\longleftrightarrow$ *matches* γ *m a p*
using *some*
apply(*simp add*: *compress-normalize-primitive-def*)
apply(*case-tac primitive-extractor* (*disc*,*sel*) *m*)
apply(*rename-tac as rst*, *simp*)
apply(*drule primitive-extractor-correct*(*1*)[*OF normalized wf*, **where** γ=γ **and** *a*=*a* **and** *p*=*p*])
apply(*elim exE conjE*)
apply(*drule f-correct*)
by (*meson matches-alist-and-alist-and' bunch-of-lemmata-about-matches*(*1*))

lemma *compress-normalize-primitive-None*:
assumes *normalized*: *normalized-nnf-match m*
and *wf*: *wf-disc-sel* (*disc*,*sel*) *C*
and *none*: *compress-normalize-primitive* (*disc*,*sel*) *C f m* = *None*
and *f-correct*: $\bigwedge$*as. f as* = *None* $\Longrightarrow$ $\neg$ *matches* γ (*alist-and* (*NegPos-map C as*)) *a p*
shows $\neg$ *matches* γ *m a p*
using *none*
apply(*simp add*: *compress-normalize-primitive-def*)
apply(*case-tac primitive-extractor* (*disc*, *sel*) *m*)
apply(*auto dest*: *primitive-extractor-correct*(*1*)[*OF assms*(*1*) *wf*] *f-correct*)
done

lemma *compress-normalize-primitive-hasdisc*:
assumes *am*: $\neg$ *has-disc disc2 m*

```
      and wf: wf-disc-sel (disc,sel) C
      and disc: (∀ a. ¬ disc2 (C a))
      and nm: normalized-nnf-match m
      and some: compress-normalize-primitive (disc,sel) C f m = Some m'
    shows normalized-nnf-match m' ∧ ¬ has-disc disc2 m'
  proof −
      from compress-normalize-primitive-nnf[OF wf nm some] have goal1: normalized-nnf-match m' .
      obtain as ms where asms: primitive-extractor (disc, sel) m = (as, ms) by fastforce
      from am primitive-extractor-correct(4)[OF nm wf asms] have 1: ¬ has-disc disc2 ms by simp
      { fix is-pos is-neg
       from disc have x1: ¬ has-disc disc2 (alist-and' (NegPos-map C (map Pos is-pos)))
       by(simp add: has-disc-alist-and' NegPos-map-map-Pos negation-type-to-match-expr-simps)
       from disc have x2: ¬ has-disc disc2 (alist-and' (NegPos-map C (map Neg is-neg)))
       by(simp add: has-disc-alist-and' NegPos-map-map-Neg negation-type-to-match-expr-simps)
        from x1 x2 have ¬ has-disc disc2 (alist-and' (NegPos-map C (map Pos is-pos @ map Neg is-neg)))
          apply(simp add: NegPos-map-append has-disc-alist-and') by blast
      }
      with some have ¬ has-disc disc2 m'
        apply(simp add: compress-normalize-primitive-def asms)
        apply(elim exE conjE)
        using 1 by fastforce
      with goal1 show ?thesis by simp
  qed
 lemma compress-normalize-primitive-hasdisc-negated:
   assumes am: ¬ has-disc-negated disc2 neg m
      and wf: wf-disc-sel (disc,sel) C
      and disc: (∀ a. ¬ disc2 (C a))
      and nm: normalized-nnf-match m
      and some: compress-normalize-primitive (disc,sel) C f m = Some m'
    shows normalized-nnf-match m' ∧ ¬ has-disc-negated disc2 neg m'
  proof −
      from compress-normalize-primitive-nnf[OF wf nm some] have goal1: normalized-nnf-match m' .
      obtain as ms where asms: primitive-extractor (disc, sel) m = (as, ms) by fastforce
     from am primitive-extractor-correct(6)[OF nm wf asms] have 1: ¬ has-disc-negated disc2 neg ms by simp
      { fix is-pos is-neg
        from disc have x1: ¬ has-disc-negated disc2 neg (alist-and' (NegPos-map C (map Pos is-pos)))
            by(simp add: has-disc-negated-alist-and' NegPos-map-map-Pos negation-type-to-match-expr-simps)
        from disc have x2: ¬ has-disc-negated disc2 neg (alist-and' (NegPos-map
```

```
C (map Neg is-neg)))
            by(simp add: has-disc-negated-alist-and' NegPos-map-map-Neg negation-type-to-match-expr-simps)
        from x1 x2 have ¬ has-disc-negated disc2 neg (alist-and' (NegPos-map C (map Pos is-pos @ map Neg is-neg)))
            apply(simp add: NegPos-map-append has-disc-negated-alist-and') by blast
      }
      with some have ¬ has-disc-negated disc2 neg m'
       apply(simp add: compress-normalize-primitive-def asms)
       apply(elim exE conjE)
       using 1 by fastforce

      with goal1 show ?thesis by simp
  qed


 thm normalize-primitive-extract-preserves-unrelated-normalized-n-primitive
 lemma compress-normalize-primitve-preserves-normalized-n-primitive:
   assumes am: normalized-n-primitive (disc2, sel2) P m
      and wf: wf-disc-sel (disc,sel) C
      and disc: (∀ a. ¬ disc2 (C a))
      and nm: normalized-nnf-match m
      and some: compress-normalize-primitive (disc,sel) C f m = Some m'
    shows normalized-nnf-match m' ∧ normalized-n-primitive (disc2, sel2) P m'
  proof −
      from compress-normalize-primitive-nnf[OF wf nm some] have goal1: normalized-nnf-match m' .
     obtain as ms where asms: primitive-extractor (disc, sel) m = (as, ms) by fastforce
        from am primitive-extractor-correct[OF nm wf asms] have 1: normalized-n-primitive (disc2, sel2) P ms by fast
      { fix iss
     from disc have normalized-n-primitive (disc2, sel2) P (alist-and (NegPos-map C iss))
         apply(induction iss)
          apply(simp-all)
         apply(rename-tac i iss, case-tac i)
          apply(simp-all)
         done
      }
      with some have normalized-n-primitive (disc2, sel2) P m'
       apply(simp add: compress-normalize-primitive-def asms)
       apply(elim exE conjE)
       using 1 normalized-n-primitive-alist-and' normalized-n-primitive-alist-and
             normalized-n-primitive.simps(4) by blast
      with goal1 show ?thesis by simp
  qed
```

24.5 Processing a list of normalization functions

fun *compress-normalize-primitive-monad* :: (*'a match-expr* ⇒ *'a match-expr option*) *list* ⇒ *'a match-expr* ⇒ *'a match-expr option* **where**
compress-normalize-primitive-monad [] *m* = *Some m* |
compress-normalize-primitive-monad (*f*#*fs*) *m* = (*case f m of None* ⇒ *None* | *Some m'* ⇒ *compress-normalize-primitive-monad fs m'*)

lemma *compress-normalize-primitive-monad*:
assumes ⋀*m m' f*. *f* ∈ *set fs* ⟹ *normalized-nnf-match m* ⟹ *f m* = *Some m'* ⟹ *matches γ m' a p* ⟷ *matches γ m a p*
and ⋀*m m' f*. *f* ∈ *set fs* ⟹ *normalized-nnf-match m* ⟹ *f m* = *Some m'* ⟹ *normalized-nnf-match m'*
and *normalized-nnf-match m*
and (*compress-normalize-primitive-monad fs m*) = *Some m'*
shows *matches γ m' a p* ⟷ *matches γ m a p* (**is** *?goal1*)
and *normalized-nnf-match m'* (**is** *?goal2*)
proof −

have *goals*: *?goal1* ∧ *?goal2*
using *assms* **proof**(*induction fs arbitrary*: *m*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons f fs*)
from *Cons.prems*(*1*) **have** *IH-prem1*:
(⋀*f m m'*. *f* ∈ *set fs* ⟹ *normalized-nnf-match m* ⟹ *f m* = *Some m'* ⟹ *matches γ m' a p* = *matches γ m a p*) **by** *auto*
from *Cons.prems*(*2*) **have** *IH-prem2*:
(⋀*f m m'*. *f* ∈ *set fs* ⟹ *normalized-nnf-match m* ⟹ *f m* = *Some m'* ⟹ *normalized-nnf-match m'*) **by** *auto*
from *Cons.IH IH-prem1 IH-prem2* **have**
IH: ⋀*m*. *normalized-nnf-match m* ⟹ *compress-normalize-primitive-monad fs m* = *Some m'* ⟹
(*matches γ m' a p* ⟷ *matches γ m a p*) ∧ *?goal2* **by** *fast*
show *?case*
proof(*cases f m*)
case *None* **thus** *?thesis* **using** *Cons.prems* **by** *auto*
next
case(*Some m''*)
from *Some Cons.prems*(*1*)[*of f*] *Cons.prems*(*3*) **have** *1*: *matches γ m'' a p* = *matches γ m a p* **by** *simp*
from *Some Cons.prems*(*2*)[*of f*] *Cons.prems*(*3*) **have** *2*: *normalized-nnf-match m''* **by** *simp*
from *Some* **have** *compress-normalize-primitive-monad* (*f* # *fs*) *m* = *compress-normalize-primitive-monad fs m''* **by** *simp*
thus *?thesis* **using** *Cons.prems*(*4*) *IH 1 2* **by** *auto*
qed
qed
from *goals* **show** *?goal1* **by** *simp*

from *goals* **show** *?goal2* **by** *simp*
qed

lemma *compress-normalize-primitive-monad-None*:
assumes $\bigwedge m\ m'\ f.\ f \in set\ fs \Longrightarrow normalized\text{-}nnf\text{-}match\ m \Longrightarrow f\ m = Some\ m' \Longrightarrow matches\ \gamma\ m'\ a\ p \longleftrightarrow matches\ \gamma\ m\ a\ p$
and $\bigwedge m\ f.\ f \in set\ fs \Longrightarrow normalized\text{-}nnf\text{-}match\ m \Longrightarrow f\ m = None \Longrightarrow \neg\ matches\ \gamma\ m\ a\ p$
and $\bigwedge m\ m'\ f.\ f \in set\ fs \Longrightarrow normalized\text{-}nnf\text{-}match\ m \Longrightarrow f\ m = Some\ m' \Longrightarrow normalized\text{-}nnf\text{-}match\ m'$
and *normalized-nnf-match m*
and (*compress-normalize-primitive-monad fs m*) = *None*
shows ¬ *matches* γ *m a p*
using *assms* **proof**(*induction fs arbitrary*: *m*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons f fs*)
from *Cons.prems*(*1*) **have** *IH-prem1*:
($\bigwedge f\ m\ m'.\ f \in set\ fs \Longrightarrow normalized\text{-}nnf\text{-}match\ m \Longrightarrow f\ m = Some\ m' \Longrightarrow matches\ \gamma\ m'\ a\ p = matches\ \gamma\ m\ a\ p$) **by** *auto*
from *Cons.prems*(*2*) **have** *IH-prem2*:
($\bigwedge f\ m\ m'.\ f \in set\ fs \Longrightarrow normalized\text{-}nnf\text{-}match\ m \Longrightarrow f\ m = None \Longrightarrow \neg\ matches\ \gamma\ m\ a\ p$) **by** *auto*
from *Cons.prems*(*3*) **have** *IH-prem3*:
($\bigwedge f\ m\ m'.\ f \in set\ fs \Longrightarrow normalized\text{-}nnf\text{-}match\ m \Longrightarrow f\ m = Some\ m' \Longrightarrow normalized\text{-}nnf\text{-}match\ m'$) **by** *auto*
from *Cons.IH IH-prem1 IH-prem2 IH-prem3* **have**
IH: $\bigwedge m.$ *normalized-nnf-match m* ⟹ *compress-normalize-primitive-monad fs m* = *None* ⟹ ¬ *matches* γ *m a p* **by** *blast*
show *?case*
proof(*cases f m*)
case *None* **thus** *?thesis* **using** *Cons.prems*(*4*) *Cons.prems*(*2*) *Cons.prems*(*3*) **by** *auto*
next
case(*Some m''*)
from *Some Cons.prems*(*3*)[*of f*] *Cons.prems*(*4*) **have** *2*: *normalized-nnf-match m''* **by** *simp*
from *Some* **have** *compress-normalize-primitive-monad* (*f* # *fs*) *m* = *compress-normalize-primitive-monad fs m''* **by** *simp*
hence ¬ *matches* γ *m'' a p* **using** *Cons.prems*(*5*) *IH 2* **by** *simp*
thus *?thesis* **using** *Cons.prems*(*1*) *Cons.prems*(*4*) *Some* **by** *auto*
qed
qed

lemma *compress-normalize-primitive-monad-preserves*:
assumes $\bigwedge m\ m'\ f.\ f \in set\ fs \Longrightarrow normalized\text{-}nnf\text{-}match\ m \Longrightarrow f\ m = Some\ m' \Longrightarrow normalized\text{-}nnf\text{-}match\ m'$

and $\bigwedge m\ m'\ f.\ f \in set\ fs \Longrightarrow normalized\text{-}nnf\text{-}match\ m \Longrightarrow P\ m \Longrightarrow f\ m = Some\ m' \Longrightarrow P\ m'$
and *normalized-nnf-match m*
and *P m*
and (*compress-normalize-primitive-monad fs m*) = *Some m'*
shows *normalized-nnf-match m'* $\wedge$ *P m'*
using *assms* **proof**(*induction fs arbitrary*: *m*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons f fs*) **thus** *?case* **by**(*simp split*: *option.split-asm*) *blast*
qed

datatype *'a match-compress* = *CannotMatch* | *MatchesAll* | *MatchExpr 'a*

end

25 Combine Match Expressions

theory *MatchExpr-Fold*
imports *Primitive-Normalization*
begin

fun *andfold-MatchExp* :: *'a match-expr list* $\Rightarrow$ *'a match-expr* **where**
andfold-MatchExp [] = *MatchAny* |
andfold-MatchExp [*e*] = *e* |
andfold-MatchExp (*e*#*es*) = *MatchAnd e* (*andfold-MatchExp es*)

lemma *andfold-MatchExp-alist-and*: *alist-and'* (*map Pos ls*) = *andfold-MatchExp* (*map Match ls*)
apply(*induction ls*)
apply(*simp*)
apply(*simp*)
apply(*rename-tac l ls*)
apply(*case-tac ls*)
by(*simp*)+

lemma *andfold-MatchExp-matches*:
matches γ (*andfold-MatchExp ms*) *a p* $\longleftrightarrow$ ($\forall$ *m* $\in$ *set ms. matches* γ *m a p*)
apply(*induction ms rule*: *andfold-MatchExp.induct*)
apply(*simp add*: *bunch-of-lemmata-about-matches*)+
done

lemma *andfold-MatchExp-not-discI*:
$\forall$ *m* $\in$ *set ms.* $\neg$ *has-disc disc m* $\Longrightarrow$ $\neg$ *has-disc disc* (*andfold-MatchExp ms*)

by(*induction ms rule*: *andfold-MatchExp.induct*) (*simp*)+

lemma *andfold-MatchExp-not-disc-negatedI*:
$\forall\, m \in$ *set ms*. $\neg$ *has-disc-negated disc neg m* $\Longrightarrow$ $\neg$ *has-disc-negated disc neg* (*andfold-MatchExp ms*)
by(*induction ms rule*: *andfold-MatchExp.induct*) (*simp*)+

lemma *andfold-MatchExp-not-disc-negated-mapMatch*:
$\neg$ *has-disc-negated disc False* (*andfold-MatchExp* (*map* (*Match* $\circ$ *C*) *ls*))
apply(*induction ls*)
apply(*simp*; *fail*)
apply(*simp*)
apply(*rename-tac ls*, *case-tac ls*)
by(*simp*)+

lemma *andfold-MatchExp-not-disc-mapMatch*:
$\forall\, a$. $\neg$ *disc* (*C a*) $\Longrightarrow$ $\neg$ *has-disc disc* (*andfold-MatchExp* (*map* (*Match* $\circ$ *C*) *ls*))
apply(*induction ls*)
apply(*simp*; *fail*)
apply(*simp*)
apply(*rename-tac ls*, *case-tac ls*)
by(*simp*)+

lemma *andfold-MatchExp-normalized-nnf*: $\forall\, m \in$ *set ms*. *normalized-nnf-match m* $\Longrightarrow$
normalized-nnf-match (*andfold-MatchExp ms*)
by(*induction ms rule*: *andfold-MatchExp.induct*)(*simp*)+

lemma *andfold-MatchExp-normalized-n-primitive*: $\forall\, m \in$ *set ms*. *normalized-n-primitive* (*disc*, *sel*) *f m* $\Longrightarrow$
normalized-n-primitive (*disc*, *sel*) *f* (*andfold-MatchExp ms*)
by(*induction ms rule*: *andfold-MatchExp.induct*)(*simp*)+

lemma *andfold-MatchExp-normalized-normalized-n-primitive-single*:
$\forall\, a$. $\neg$ *disc* (*C a*) $\Longrightarrow$
s $\in$ *set* (*normalize-match* (*andfold-MatchExp* (*map* (*Match* $\circ$ *C*) *xs*))) $\Longrightarrow$
normalized-n-primitive (*disc*, *sel*) *f s*
apply(*rule normalized-n-primitive-if-no-primitive*)
using *normalized-nnf-match-normalize-match* **apply** *blast*
apply(*rule normalize-match-preserves-nodisc*[**where** *m*=(*andfold-MatchExp* (*map* (*Match* $\circ$ *C*) *xs*))])
apply *simp-all*
by (*simp add*: *andfold-MatchExp-not-discI*)

lemma *normalize-andfold-MatchExp-normalized-n-primitive*:
$\forall$ *m* $\in$ *set ms*. $\forall$ *s'* $\in$ *set* (*normalize-match m*). *normalized-n-primitive* (*disc*, *sel*) *f s'* $\Longrightarrow$
s $\in$ *set* (*normalize-match* (*andfold-MatchExp ms*)) $\Longrightarrow$
normalized-n-primitive (*disc*, *sel*) *f s*

proof(*induction ms arbitrary*: *s rule*: *andfold-MatchExp.induct*)
case *1* **thus** *?case* **by** *simp*
next
case *2* **thus** *?case* **by** *simp*
next
case (*3 v1 v2 va*)
have *IH*: $s' \in$ *set* (*normalize-match* (*andfold-MatchExp* (*v2* # *va*))) $\Longrightarrow$
normalized-n-primitive (*disc*, *sel*) *f s'* **for** *s'*
using *3*(*1*)[*of s'*]
apply(*simp*)
using *3*(*2*) **by** *force*
from *3*(*2*,*3*) *IH* **show** *?case* **by**(*clarsimp*)
qed
end
theory *Common-Primitive-Lemmas*
imports *Common-Primitive-Matcher*
../Semantics-Ternary/Primitive-Normalization
../Semantics-Ternary/MatchExpr-Fold
begin

26 Further Lemmas about the Common Matcher

lemma *has-unknowns-common-matcher*: **fixes** *m*::'*i*::*len common-primitive match-expr*
shows *has-unknowns common-matcher m* $\longleftrightarrow$ *has-disc is-Extra m*
proof −
{ fix *A* **and** *p* :: ('*i*, '*a*) *tagged-packet-scheme*
have *common-matcher A p* = *TernaryUnknown* $\longleftrightarrow$ *is-Extra A*
by(*induction A p rule*: *common-matcher.induct*) (*simp-all add*: *bool-to-ternary-Unknown*)
} hence β = (*common-matcher*::('*i*::*len common-primitive*, ('*i*, '*a*) *tagged-packet-scheme*)
exact-match-tac)
$\Longrightarrow$ *has-unknowns* β *m* = *has-disc is-Extra m* **for** β
by(*induction* β *m rule*: *has-unknowns.induct*)
(*simp-all*)
thus *?thesis* **by** *simp*
qed

end
theory *Ports-Normalize*
imports *Common-Primitive-Lemmas*
begin

27 Normalizing L4 Ports

27.1 Defining Normalized Ports

fun *normalized-src-ports* :: '*i*::*len common-primitive match-expr* ⇒ *bool* **where**

normalized-src-ports MatchAny = True |
normalized-src-ports (Match (Src-Ports (L4Ports - []))) = True |
normalized-src-ports (Match (Src-Ports (L4Ports - [-]))) = True |
normalized-src-ports (Match (Src-Ports -)) = False |
normalized-src-ports (Match -) = True |
normalized-src-ports (MatchNot (Match (Src-Ports -))) = False |
normalized-src-ports (MatchNot (Match -)) = True |
normalized-src-ports (MatchAnd m1 m2) = (normalized-src-ports m1 ∧ normalized-src-ports m2) |
normalized-src-ports (MatchNot (MatchAnd - -)) = False |
normalized-src-ports (MatchNot (MatchNot -)) = False |
normalized-src-ports (MatchNot MatchAny) = True

fun *normalized-dst-ports* :: *'i::len common-primitive match-expr ⇒ bool* **where**
normalized-dst-ports MatchAny = True |
normalized-dst-ports (Match (Dst-Ports (L4Ports - []))) = True |
normalized-dst-ports (Match (Dst-Ports (L4Ports - [-]))) = True |
normalized-dst-ports (Match (Dst-Ports -)) = False |
normalized-dst-ports (Match -) = True |
normalized-dst-ports (MatchNot (Match (Dst-Ports -))) = False |
normalized-dst-ports (MatchNot (Match -)) = True |
normalized-dst-ports (MatchAnd m1 m2) = (normalized-dst-ports m1 ∧ normalized-dst-ports m2) |
normalized-dst-ports (MatchNot (MatchAnd - -)) = False |
normalized-dst-ports (MatchNot (MatchNot -)) = False |
normalized-dst-ports (MatchNot MatchAny) = True

lemma *normalized-src-ports-def2*: *normalized-src-ports ms = normalized-n-primitive (is-Src-Ports, src-ports-sel) (λps. case ps of L4Ports - pts ⇒ length pts ≤ 1) ms*
by(*induction ms rule*: *normalized-src-ports.induct*, *simp-all*)
lemma *normalized-dst-ports-def2*: *normalized-dst-ports ms = normalized-n-primitive (is-Dst-Ports, dst-ports-sel) (λps. case ps of L4Ports - pts ⇒ length pts ≤ 1) ms*
by(*induction ms rule*: *normalized-dst-ports.induct*, *simp-all*)

Idea: first, remove all negated matches, then *normalize-match*, then only work with *primitive-extractor* on *Pos* ones. They only need an intersect and split later on.

This is not very efficient because normalizing nnf will blow up a lot. but we can tune performance later on go for correctness first! Anything with *MatchOr* and *normalize-match* later is a bit inefficient.

27.2 Compressing Positive Matches on Ports into a Single Match

fun *l4-ports-compress* :: *ipt-l4-ports list ⇒ ipt-l4-ports match-compress* **where**
l4-ports-compress [] = MatchesAll |
l4-ports-compress [L4Ports proto ps] = MatchExpr (L4Ports proto (wi2l (wordinterval-compress (l2wi ps)))) |

```
  l4-ports-compress (L4Ports proto1 ps1 # L4Ports proto2 ps2 # pss) =
    (if
        proto1 ≠ proto2
     then
       CannotMatch
     else
        l4-ports-compress (L4Ports proto1 (wi2l (wordinterval-intersection (l2wi
ps1) (l2wi ps2))) # pss)
    )

  value[code] l4-ports-compress [L4Ports TCP [(22,22), (23,23)]]


  lemma raw-ports-compress-src-CannotMatch:
  fixes p :: ('i::len, 'a) tagged-packet-scheme
  assumes generic: primitive-matcher-generic β
  and c: l4-ports-compress pss = CannotMatch
  shows ¬ matches (β, α) (alist-and (map (Pos ∘ Src-Ports) pss)) a p
  using c apply(induction pss rule: l4-ports-compress.induct)
    apply(simp; fail)
   apply(simp; fail)
 apply(simp add: primitive-matcher-generic.Ports-single[OF generic] bunch-of-lemmata-about-matches
split: if-split-asm)
   apply meson
  by(simp add: l2wi-wi2l ports-to-set-wordinterval)

  lemma raw-ports-compress-dst-CannotMatch:
  fixes p :: ('i::len, 'a) tagged-packet-scheme
  assumes generic: primitive-matcher-generic β
  and c: l4-ports-compress pss = CannotMatch
  shows ¬ matches (β, α) (alist-and (map (Pos ∘ Dst-Ports) pss)) a p
  using c apply(induction pss rule: l4-ports-compress.induct)
    apply(simp; fail)
   apply(simp; fail)
 apply(simp add: primitive-matcher-generic.Ports-single[OF generic] bunch-of-lemmata-about-matches
split: if-split-asm)
   apply meson
  by(simp add: l2wi-wi2l ports-to-set-wordinterval)

  lemma l4-ports-compress-length-Matchall: length pss > 0 ⟹ l4-ports-compress
pss ≠ MatchesAll
    by(induction pss rule: l4-ports-compress.induct) simp+

  lemma raw-ports-compress-MatchesAll:
  fixes p :: ('i::len, 'a) tagged-packet-scheme
  assumes generic: primitive-matcher-generic β
  and c: l4-ports-compress pss = MatchesAll
  shows matches (β, α) (alist-and (map (Pos ∘ Src-Ports) pss)) a p
  and matches (β, α) (alist-and (map (Pos ∘ Dst-Ports) pss)) a p
```

using *c* **apply**(*induction pss rule*: *l4-ports-compress.induct*)
by(*simp add*: *l4-ports-compress-length-Matchall bunch-of-lemmata-about-matches split*: *if-split-asm*)+

lemma *raw-ports-compress-src-MatchExpr*:
fixes *p* :: ('*i*::*len*, '*a*) *tagged-packet-scheme*
assumes *generic*: *primitive-matcher-generic* β
and *c*: *l4-ports-compress pss* = *MatchExpr m*
shows *matches* (β, α) (*Match* (*Src-Ports m*)) *a p* ⟷ *matches* (β, α) (*alist-and* (*map* (*Pos* ∘ *Src-Ports*) *pss*)) *a p*
using *c* **apply**(*induction pss arbitrary*: *m rule*: *l4-ports-compress.induct*)
apply(*simp add*: *bunch-of-lemmata-about-matches*; *fail*)
subgoal
apply(*simp add*: *bunch-of-lemmata-about-matches*)
apply(*drule sym*, *simp*)
by(*simp add*: *primitive-matcher-generic.Ports-single*[*OF generic*] *wordinterval-compress l2wi-wi2l ports-to-set-wordinterval*)
apply(*case-tac m*)
apply(*simp add*: *bunch-of-lemmata-about-matches split*: *if-split-asm*)
apply(*simp add*: *primitive-matcher-generic.Ports-single*[*OF generic*])
apply(*simp add*: *l2wi-wi2l ports-to-set-wordinterval*)
by *fastforce*

lemma *raw-ports-compress-dst-MatchExpr*:
fixes *p* :: ('*i*::*len*, '*a*) *tagged-packet-scheme*
assumes *generic*: *primitive-matcher-generic* β
and *c*: *l4-ports-compress pss* = *MatchExpr m*
shows *matches* (β, α) (*Match* (*Dst-Ports m*)) *a p* ⟷ *matches* (β, α) (*alist-and* (*map* (*Pos* ∘ *Dst-Ports*) *pss*)) *a p*
using *c* **apply**(*induction pss arbitrary*: *m rule*: *l4-ports-compress.induct*)
apply(*simp add*: *bunch-of-lemmata-about-matches*; *fail*)
subgoal
apply(*simp add*: *bunch-of-lemmata-about-matches*)
apply(*drule sym*, *simp*)
by(*simp add*: *primitive-matcher-generic.Ports-single*[*OF generic*] *wordinterval-compress l2wi-wi2l ports-to-set-wordinterval*)
apply(*case-tac m*)
apply(*simp add*: *bunch-of-lemmata-about-matches split*: *if-split-asm*)
apply(*simp add*: *primitive-matcher-generic.Ports-single*[*OF generic*])
apply(*simp add*: *l2wi-wi2l ports-to-set-wordinterval*)
by *fastforce*

27.3 Rewriting Negated Matches on Ports

fun *l4-ports-negate-one*
:: (*ipt-l4-ports* ⇒ '*i common-primitive*) ⇒ *ipt-l4-ports* ⇒ ('*i*::*len common-primitive*) *match-expr*
where
l4-ports-negate-one C (*L4Ports proto pts*) = *MatchOr*

(*MatchNot* (*Match* (*Prot* (*Proto proto*))))
(*Match* (*C* (*L4Ports proto* (*raw-ports-invert pts*))))

lemma *l4-ports-negate-one*:
fixes *p* :: ('*i*::*len*, '*a*) *tagged-packet-scheme*
assumes *generic*: *primitive-matcher-generic* β
shows *matches* (β, α) (*l4-ports-negate-one Src-Ports ports*) *a p* ⟷
matches (β, α) (*MatchNot* (*Match* (*Src-Ports ports*))) *a p*
and *matches* (β, α) (*l4-ports-negate-one Dst-Ports ports*) *a p* ⟷
matches (β, α) (*MatchNot* (*Match* (*Dst-Ports ports*))) *a p*
apply(*case-tac* [!] *ports*)
by(*auto simp add*: *primitive-matcher-generic.Ports-single-not*[*OF generic*]
MatchOr bunch-of-lemmata-about-matches
primitive-matcher-generic.Prot-single-not[*OF generic*]
primitive-matcher-generic.Ports-single[*OF generic*]
raw-ports-invert)

lemma *l4-ports-negate-one-nodisc*:
∀ *a*. ¬ *disc* (*C a*) ⟹ ∀ *a*. ¬ *disc* (*Prot a*) ⟹ ¬ *has-disc disc* (*l4-ports-negate-one C pt*)
apply(*cases pt*)
by(*simp add*: *MatchOr-def*)

lemma *l4-ports-negate-one-not-has-disc-negated-generic*:
assumes *noProt*: ∀ *a*. ¬ *disc* (*Prot a*)
shows ¬ *has-disc-negated disc False* (*l4-ports-negate-one C ports*)
apply(*cases ports*, *rename-tac proto pts*)
by(*simp add*: *MatchOr-def noProt*)

lemma *l4-ports-negate-one-not-has-disc-negated*:
¬ *has-disc-negated is-Src-Ports False* (*l4-ports-negate-one Src-Ports ports*)
¬ *has-disc-negated is-Dst-Ports False* (*l4-ports-negate-one Dst-Ports ports*)
by(*simp add*: *l4-ports-negate-one-not-has-disc-negated-generic*)+

lemma *negated-normalized-folded-ports-nodisc*:
∀ *a*. ¬ *disc* (*C a*) ⟹ (∀ *a*. ¬ *disc* (*Prot a*)) ∨ *pts* = [] ⟹
m ∈ *set* (*normalize-match* (*andfold-MatchExp* (*map* (*l4-ports-negate-one C*) *pts*))) ⟹
¬ *has-disc disc m*
apply(*subgoal-tac* ¬ *has-disc disc* (*andfold-MatchExp* (*map* (*l4-ports-negate-one C*) *pts*)))
prefer *2*
apply(*rule andfold-MatchExp-not-discI*)
apply(*simp*)
apply(*elim disjE*)
using *l4-ports-negate-one-nodisc* **apply** *blast*
apply(*simp*; *fail*)
using *normalize-match-preserves-nodisc* **by** *blast*

lemma *negated-normalized-folded-ports-normalized-n-primitive*:
∀ *a*. ¬ *disc* (*C a*) ⟹ (∀ *a*. ¬ *disc* (*Prot a*)) ∨ *pts* = [] ⟹
x ∈ *set* (*normalize-match* (*andfold-MatchExp* (*map* (*l4-ports-negate-one C*) *pts*))) ⟹
normalized-n-primitive (*disc*, *sel*) *f x*
apply(*rule normalized-n-primitive-if-no-primitive*)
using *normalized-nnf-match-normalize-match* **apply** *blast*
apply(*rule negated-normalized-folded-ports-nodisc*)
by *simp-all*

beware, the result is not nnf normalized!

lemma ¬ *normalized-nnf-match* (*l4-ports-negate-one C ports*)
by(*cases ports*) (*simp add*: *MatchOr-def*)

Warning: does not preserve negated primitive property in general. Might be violated for *Prot*. We will nnf normalize after applying the function.

lemma ∀ *a*. ¬ *disc* (*C a*) ⟹ ¬ *normalized-n-primitive* (*disc*, *sel*) *f* (*l4-ports-negate-one C a*)
by(*cases a*)(*simp add*: *MatchOr-def*)

declare *l4-ports-negate-one.simps*[*simp del*]

lemma ((*normalize-match* (*l4-ports-negate-one Src-Ports* (*L4Ports TCP* [(*22*,*22*),(*80*,*90*)])))::*32 common-primitive match-expr list*)
=
[*MatchNot* (*Match* (*Prot* (*Proto TCP*)))
, *Match* (*Src-Ports* (*L4Ports 6* [(*0*, *21*), (*23*, *79*), (*91*, *0xFFFF*)]))] **by** *eval*

definition *rewrite-negated-primitives*
:: ((′*a* ⇒ *bool*) × (′*a* ⇒ ′*b*)) ⇒ (′*b* ⇒ ′*a*) ⇒ — *disc-sel C*
((′*b* ⇒ ′*a*) ⇒ ′*b* ⇒ ′*a match-expr*) ⇒ — *negate-one* function
′*a match-expr* ⇒ ′*a match-expr* **where**
rewrite-negated-primitives disc-sel C negate m ≡
let (*spts*, *rst*) = *primitive-extractor disc-sel m*
in if getNeg spts = [] *then m else*
MatchAnd
(*andfold-MatchExp* (*map* (*negate C*) (*getNeg spts*)))
(*MatchAnd*
(*andfold-MatchExp* (*map* (*Match* ∘ *C*) (*getPos spts*))) — TODO: compress all the positive ports into one?
rst)

It does nothing of there is not even a negated primitive in it

lemma *rewrite-negated-primitives-unchanged-if-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m*
and *wf-disc-sel*: *wf-disc-sel* (*disc*,*sel*) *C*
and *noDisc*: ¬ *has-disc-negated disc False m*

shows *rewrite-negated-primitives* (*disc*,*sel*) *C negate-f m* = *m*
apply(*simp add*: *rewrite-negated-primitives-def*)
apply(*case-tac primitive-extractor* (*disc*,*sel*) *m*, *rename-tac spts rst*)
apply(*simp*)
apply(*frule primitive-extractor-correct*(*8*)[*OF n wf-disc-sel*])
using *noDisc* **by** *blast*

lemma *rewrite-negated-primitives-normalized-no-modification*:
assumes *wf-disc-sel*: *wf-disc-sel* (*disc*, *sel*) *C*
and *disc-p*: ¬ *has-disc-negated disc False m*
and *n*: *normalized-nnf-match m*
and *a*: *a* ∈ *set* (*normalize-match* (*rewrite-negated-primitives* (*disc*, *sel*) *C l4-ports-negate-one m*))
shows *a* = *m*
proof −
from *rewrite-negated-primitives-unchanged-if-not-has-disc-negated*[*OF n wf-disc-sel disc-p*]
have *m*: *rewrite-negated-primitives* (*disc*, *sel*) *C l4-ports-negate-one m* = *m* **by** *simp*
from *a* **show** *?thesis*
apply(*subst*(*asm*) *m*)
using *normalize-match-already-normalized*[*OF n*] **by** *fastforce*
qed

lemma *rewrite-negated-primitives-preserves-not-has-disc*:
assumes *n*: *normalized-nnf-match m*
and *wf-disc-sel*: *wf-disc-sel* (*disc*, *sel*) *C*
and *nodisc*: ¬ *has-disc disc2 m*
and *noNeg*: ¬ *has-disc-negated disc False m*
and *disc2-noC*: ∀ *a*. ¬ *disc2* (*C a*)
shows ¬ *has-disc disc2* (*rewrite-negated-primitives* (*disc*, *sel*) *C l4-ports-negate-one m*)
apply(*subst rewrite-negated-primitives-unchanged-if-not-has-disc-negated*)
using *n wf-disc-sel noNeg nodisc* **by**(*simp*)+

lemma *rewrite-negated-primitives*:
assumes *n*: *normalized-nnf-match m* **and** *wf-disc-sel*: *wf-disc-sel disc-sel C*
and *negate-f*: ∀ *pts*. *matches* γ (*negate-f C pts*) *a p* ⟷ *matches* γ (*MatchNot* (*Match* (*C pts*))) *a p*
shows *matches* γ (*rewrite-negated-primitives disc-sel C negate-f m*) *a p* ⟷ *matches* γ *m a p*
proof −
obtain *spts rst* **where** *pext*: *primitive-extractor disc-sel m* = (*spts*, *rst*)
by(*cases primitive-extractor disc-sel m*) *simp*
obtain *disc sel* **where** *disc-sel*: *disc-sel* = (*disc*, *sel*) **by**(*cases disc-sel*) *simp*
with *wf-disc-sel* **have** *wf-disc-sel′*: *wf-disc-sel* (*disc*, *sel*) *C* **by** *simp*
from *disc-sel pext* **have** *pext′*: *primitive-extractor* (*disc*, *sel*) *m* = (*spts*, *rst*) **by** *simp*

have *matches* γ (*andfold-MatchExp* (*map* (*negate-f C*) (*getNeg spts*))) *a p* $\wedge$
matches γ (*andfold-MatchExp* (*map* (*Match* $\circ$ *C*) (*getPos spts*))) *a p* $\wedge$
matches γ *rst a p* $\longleftrightarrow$
matches γ *m a p*
apply(*subst primitive-extractor-correct*(*1*)[*OF n wf-disc-sel' pext'*, *symmetric*])
apply(*simp add*: *andfold-MatchExp-matches*)
apply(*simp add*: *negate-f*)
using *alist-and-NegPos-map-getNeg-getPos-matches* **by** *fast*
thus *?thesis* **by**(*simp add*: *rewrite-negated-primitives-def pext bunch-of-lemmata-about-matches*)
qed

lemma *rewrite-negated-primitives-not-has-disc*:
assumes *n*: *normalized-nnf-match m* **and** *wf-disc-sel*: *wf-disc-sel* (*disc*,*sel*) *C*
and *nodisc*: $\neg$ *has-disc disc2 m*

and *negate-f*: *has-disc-negated disc False m* $\Longrightarrow$ $\forall$ *pts*. $\neg$ *has-disc disc2* (*negate-f C pts*)
and *no-disc*: $\forall$ *a*. $\neg$ *disc2* (*C a*)
shows $\neg$ *has-disc disc2* (*rewrite-negated-primitives* (*disc*,*sel*) *C negate-f m*)
apply(*simp add*: *rewrite-negated-primitives-def*)
apply(*case-tac primitive-extractor* (*disc*,*sel*) *m*, *rename-tac spts rst*)
apply(*simp*)
apply(*frule primitive-extractor-correct*(*4*)[*OF n wf-disc-sel*])
apply(*frule primitive-extractor-correct*(*8*)[*OF n wf-disc-sel*])
apply(*intro conjI impI*)
using *nodisc* **apply**(*simp*; *fail*)
apply(*rule andfold-MatchExp-not-discI*)
apply(*simp add*: *negate-f*; *fail*)
using *andfold-MatchExp-not-disc-mapMatch no-disc* **apply** *blast*
using *nodisc* **by** *blast*

lemma *rewrite-negated-primitives-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m* **and** *wf-disc-sel*: *wf-disc-sel* (*disc*,*sel*) *C*
and *negate-f*: *has-disc-negated disc False m* $\Longrightarrow$ $\forall$ *pts*. $\neg$ *has-disc-negated disc False* (*negate-f C pts*)
shows $\neg$ *has-disc-negated disc False* (*rewrite-negated-primitives* (*disc*,*sel*) *C negate-f m*)
apply(*simp add*: *rewrite-negated-primitives-def*)
apply(*case-tac primitive-extractor* (*disc*,*sel*) *m*, *rename-tac spts rst*)
apply(*simp*)
apply(*frule primitive-extractor-correct*(*3*)[*OF n wf-disc-sel*])
apply(*frule primitive-extractor-correct*(*8*)[*OF n wf-disc-sel*])
apply(*intro conjI impI*)
apply *blast*
apply(*rule andfold-MatchExp-not-disc-negatedI*)
apply(*simp add*: *negate-f*; *fail*)
using *andfold-MatchExp-not-disc-negated-mapMatch* **apply** *blast*
using *has-disc-negated-has-disc* **by** *blast*

lemma *rewrite-negated-primitives-preserves-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m* **and** *wf-disc-sel*: *wf-disc-sel* (*disc*,*sel*) *C*
and *negate-f*: *has-disc-negated disc False m* $\Longrightarrow$ $\forall$ *pts*. $\neg$ *has-disc-negated disc2 False* (*negate-f C pts*)
and *no-disc*: $\neg$ *has-disc-negated disc2 False m*
shows $\neg$ *has-disc-negated disc2 False* (*rewrite-negated-primitives* (*disc*,*sel*) *C negate-f m*)
apply(*simp add*: *rewrite-negated-primitives-def*)
apply(*case-tac primitive-extractor* (*disc*,*sel*) *m*, *rename-tac spts rst*)
apply(*simp*)
apply(*frule primitive-extractor-correct*(*3*)[*OF n wf-disc-sel*])
apply(*frule primitive-extractor-correct*(*8*)[*OF n wf-disc-sel*])
apply(*intro conjI impI*)
using *no-disc* **apply** *blast*
apply(*rule andfold-MatchExp-not-disc-negatedI*)
apply(*simp add*: *negate-f*; *fail*)
using *andfold-MatchExp-not-disc-negated-mapMatch* **apply** *blast*
apply(*drule primitive-extractor-correct*(*6*)[*OF n wf-disc-sel*, **where** *neg*=*False*])
using *no-disc* **by** *blast*

lemma *rewrite-negated-primitives-normalized-preserves-unrelated-helper*:
assumes *wf-disc-sel*: *wf-disc-sel* (*disc*, *sel*) *C*
and *disc*: $\forall$ *a*. $\neg$ *disc2* (*C a*)
and *disc-p*: ($\forall$ *a*. $\neg$ *disc2* (*Prot a*)) $\vee$ $\neg$ *has-disc-negated disc False m*
shows *normalized-nnf-match m* $\Longrightarrow$
normalized-n-primitive (*disc2*, *sel2*) *f m* $\Longrightarrow$
a $\in$ *set* (*normalize-match* (*rewrite-negated-primitives* (*disc*, *sel*) *C l4-ports-negate-one m*)) $\Longrightarrow$
normalized-n-primitive (*disc2*, *sel2*) *f a*
proof $-$
have *helper-a-normalized*: *a* $\in$ *MatchAnd x* ‘ ($\bigcup$ *x*$\in$*set spts*. *MatchAnd x* ‘ *set* (*normalize-match rst*)) $\Longrightarrow$
normalized-n-primitive (*disc*, *sel*) *f x* $\Longrightarrow$
($\forall$ *s* $\in$ *set spts*. *normalized-n-primitive* (*disc*, *sel*) *f s*) $\Longrightarrow$
normalized-n-primitive (*disc*, *sel*) *f rst* $\Longrightarrow$
normalized-n-primitive (*disc*, *sel*) *f a*
for *a x spts rst f disc* **and** *sel*::*'a common-primitive* $\Rightarrow$ *'b*
apply(*subgoal-tac* $\exists$ *s r*. *a* = *MatchAnd x* (*MatchAnd s r*) $\wedge$ *s* $\in$ *set spts* $\wedge$ *r* $\in$ *set* (*normalize-match rst*))
prefer *2*
apply *blast*
apply(*elim exE conjE*, *rename-tac s r*)
apply(*simp*)
using *normalize-match-preserves-normalized-n-primitive* **by** *blast*

show *normalized-nnf-match m* $\Longrightarrow$
normalized-n-primitive (*disc2*, *sel2*) *f m* $\Longrightarrow$

```
      a ∈ set (normalize-match (rewrite-negated-primitives (disc, sel) C l4-ports-negate-one
m)) ⟹
        normalized-n-primitive (disc2, sel2) f  a
    apply(case-tac ¬ has-disc-negated disc False m)
     subgoal
       using rewrite-negated-primitives-normalized-no-modification[OF wf-disc-sel]
by blast
    apply(simp add: rewrite-negated-primitives-def)
    apply(case-tac primitive-extractor (disc, sel) m, rename-tac spts rst)
    apply(simp)
    apply(subgoal-tac normalized-n-primitive (disc2, sel2) f rst)
     prefer 2 subgoal for spts rst
     apply(drule primitive-extractor-correct(5)[OF - wf-disc-sel, where P=f])
      apply blast
     by(simp)
    apply(insert disc-p, simp)
    apply(drule(1) primitive-extractor-correct(8)[OF - wf-disc-sel])
    apply(simp)
    apply(elim bexE)
    apply(erule helper-a-normalized)
      subgoal for spts
    apply(rule-tac pts=(getNeg spts) in negated-normalized-folded-ports-normalized-n-primitive[where
C=C])
       using disc apply(simp; fail)
      using disc-p primitive-extractor-correct(8)[OF - wf-disc-sel] apply blast
     by simp
     subgoal for x
     apply(intro ballI)
    apply(rule andfold-MatchExp-normalized-normalized-n-primitive-single[where
C=C])
      using disc disc-p by(simp)+
   by blast
  qed


  definition rewrite-negated-src-ports
     :: 'i::len common-primitive match-expr ⇒ 'i common-primitive match-expr
where
    rewrite-negated-src-ports m ≡
       rewrite-negated-primitives (is-Src-Ports, src-ports-sel) Src-Ports l4-ports-negate-one
m

  definition rewrite-negated-dst-ports
     :: 'i::len common-primitive match-expr ⇒ 'i common-primitive match-expr
where
    rewrite-negated-dst-ports m ≡
       rewrite-negated-primitives (is-Dst-Ports, dst-ports-sel) Dst-Ports l4-ports-negate-one
m
```

value *rewrite-negated-src-ports* (*MatchAnd* (*Match* (*Dst* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*127*, *0*, *0*, *0*)) *8*)))
(*MatchAnd* (*Match* (*Prot* (*Proto TCP*)))
(*MatchNot* (*Match* (*Src-Ports* (*L4Ports UDP* [(*80*,*80*)])))))
))
value *rewrite-negated-src-ports* (*MatchAnd* (*Match* (*Dst* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*127*, *0*, *0*, *0*)) *8*)))
(*MatchAnd* (*Match* (*Prot* (*Proto TCP*)))
(*MatchNot* (*Match* (*Extra* ''*foobar*''))))
))

lemma *rewrite-negated-src-ports*:
assumes *generic*: *primitive-matcher-generic* β **and** *n*: *normalized-nnf-match m*
shows *matches* (β, α) (*rewrite-negated-src-ports m*) *a p* $\longleftrightarrow$ *matches* (β, α) *m a p*
apply(*simp add*: *rewrite-negated-src-ports-def*)
apply(*rule rewrite-negated-primitives*)
by(*simp add*: *l4-ports-negate-one*[*OF generic*] *n wf-disc-sel-common-primitive*(*1*))+

lemma *rewrite-negated-dst-ports*:
assumes *generic*: *primitive-matcher-generic* β **and** *n*: *normalized-nnf-match m*
shows *matches* (β, α) (*rewrite-negated-dst-ports m*) *a p* $\longleftrightarrow$ *matches* (β, α) *m a p*
apply(*simp add*: *rewrite-negated-dst-ports-def*)
apply(*rule rewrite-negated-primitives*)
by(*simp add*: *l4-ports-negate-one*[*OF generic*] *n wf-disc-sel-common-primitive*(*2*))+

lemma *rewrite-negated-src-ports-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m*
shows $\neg$ *has-disc-negated is-Src-Ports False* (*rewrite-negated-src-ports m*)
apply(*simp add*: *rewrite-negated-src-ports-def*)
apply(*rule rewrite-negated-primitives-not-has-disc-negated*)
by(*simp add*: *n wf-disc-sel-common-primitive*(*1*) *l4-ports-negate-one-not-has-disc-negated*)+

lemma *rewrite-negated-dst-ports-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m*
shows $\neg$ *has-disc-negated is-Dst-Ports False* (*rewrite-negated-dst-ports m*)
apply(*simp add*: *rewrite-negated-dst-ports-def*)
apply(*rule rewrite-negated-primitives-not-has-disc-negated*)
by(*simp add*: *n wf-disc-sel-common-primitive*(*2*) *l4-ports-negate-one-not-has-disc-negated*)+

lemma $\neg$ *has-disc-negated disc t m* $\Longrightarrow$ $\forall\, m' \in$ *set* (*normalize-match m*). $\neg$ *has-disc-negated disc t m'*
by(*fact i-m-giving-this-a-funny-name-so-i-can-thank-my-future-me-when-sledgehammer-will-find-this-one-da*

corollary *normalize-rewrite-negated-src-ports-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m*

shows $\forall\, m' \in set\ (normalize\text{-}match\ (rewrite\text{-}negated\text{-}src\text{-}ports\ m)).\ \neg\ has\text{-}disc\text{-}negated\ is\text{-}Src\text{-}Ports\ False\ m'$
apply(*rule i-m-giving-this-a-funny-name-so-i-can-thank-my-future-me-when-sledgehammer-will-find-this-one*
apply(*rule rewrite-negated-src-ports-not-has-disc-negated*)
using *n* **by** *simp*

27.4 Normalizing Positive Matches on Ports

fun *singletonize-L4Ports* :: *ipt-l4-ports* ⇒ *ipt-l4-ports list* **where**
singletonize-L4Ports (*L4Ports proto pts*) = *map* (λ*p*. *L4Ports proto* [*p*]) *pts*

lemma *singletonize-L4Ports-src*: **assumes** *generic*: *primitive-matcher-generic* β
shows *match-list* (β, α) (*map* (*Match* ∘ *Src-Ports*) (*singletonize-L4Ports pts*)) *a p* ⟷
matches (β, α) (*Match* (*Src-Ports pts*)) *a p*
apply(*cases pts*)
apply(*simp add*: *match-list-matches primitive-matcher-generic.Ports-single*[*OF generic*])
apply(*simp add*: *ports-to-set*)
by *auto*

lemma *singletonize-L4Ports-dst*: **assumes** *generic*: *primitive-matcher-generic* β
shows *match-list* (β, α) (*map* (*Match* ∘ *Dst-Ports*) (*singletonize-L4Ports pts*)) *a p* ⟷
matches (β, α) (*Match* (*Dst-Ports pts*)) *a p*
apply(*cases pts*)
apply(*simp add*: *match-list-matches primitive-matcher-generic.Ports-single*[*OF generic*])
apply(*simp add*: *ports-to-set*)
by *auto*

lemma *singletonize-L4Ports-normalized-generic*:
assumes *wf-disc-sel*: *wf-disc-sel* (*disc*,*sel*) *C*
and $m' \in (\lambda spt.\ Match\ (C\ spt))\ `\ set\ (singletonize\text{-}L4Ports\ pt)$
shows *normalized-n-primitive* (*disc*, *sel*) (*case-ipt-l4-ports* (λ*x pts*. *length pts* ≤ *1*)) *m'*
using *assms*
apply(*case-tac pt*)
apply(*simp*)
apply(*induction m'*)
by(*auto simp*: *wf-disc-sel.simps*)

lemma *singletonize-L4Ports-normalized-src-ports*:
$m' \in (\lambda spt.\ Match\ (Src\text{-}Ports\ spt))\ `\ set\ (singletonize\text{-}L4Ports\ pt) \Longrightarrow normalized\text{-}src\text{-}ports\ m'$
apply(*simp add*: *normalized-src-ports-def2*)
using *singletonize-L4Ports-normalized-generic*[*OF wf-disc-sel-common-primitive*(*1*)]
by *blast*

lemma *singletonize-L4Ports-normalized-dst-ports*:
m′ ∈ (*λspt. Match* (*Dst-Ports spt*)) ‘ *set* (*singletonize-L4Ports pt*) ⟹ *normalized-dst-ports m′*
apply(*simp add*: *normalized-dst-ports-def2*)
using *singletonize-L4Ports-normalized-generic*[*OF wf-disc-sel-common-primitive*(*2*)]
by *blast*

declare *singletonize-L4Ports.simps*[*simp del*]

lemma *normalized-ports-singletonize-combine-rst*:
assumes *wf-disc-sel*: *wf-disc-sel* (*disc*,*sel*) *C*
shows *normalized-n-primitive* (*disc*, *sel*) (*case-ipt-l4-ports* (*λx pts. length pts* ≤ *1*)) *rst* ⟹
m′ ∈ (*λspt. MatchAnd* (*Match* (*C spt*)) *rst*) ‘ *set* (*singletonize-L4Ports pt*) ⟹
normalized-n-primitive (*disc*, *sel*) (*case-ipt-l4-ports* (*λx pts. length pts* ≤ *1*)) *m′*
apply *simp*
apply(*rule normalized-n-primitive-MatchAnd-combine-map*)
apply(*simp-all*)
using *singletonize-L4Ports-normalized-generic*[*OF wf-disc-sel*] **by** *fastforce*

Normalizing match expressions such that at most one port will exist in it. Returns a list of match expressions (splits one firewall rule into several rules).

definition *normalize-positive-ports-step*
:: ((*′i*::*len common-primitive* ⇒ *bool*) × (*′i common-primitive* ⇒ *ipt-l4-ports*)) ⇒
(*ipt-l4-ports* ⇒ *′i common-primitive*) ⇒
′i common-primitive match-expr ⇒ *′i common-primitive match-expr list*
where
normalize-positive-ports-step disc-sel C m ≡
let (*spts*, *rst*) = *primitive-extractor disc-sel m in*
case (*getPos spts*, *getNeg spts*)
of (*pspts*, []) ⇒ (*case l4-ports-compress pspts of CannotMatch* ⇒ []
| *MatchesAll* ⇒ [*rst*]
| *MatchExpr m* ⇒ *map* (*λspt.* (*MatchAnd* (*Match* (*C spt*)) *rst*)) (*singletonize-L4Ports m*)
)
| (-, -) ⇒ *undefined*

lemma *normalize-positive-ports-step-nnf*:
assumes *n*: *normalized-nnf-match m* **and** *wf-disc-sel*: *wf-disc-sel* (*disc*,*sel*) *C*
and *noneg*: ¬ *has-disc-negated disc False m*
shows *m′* ∈ *set* (*normalize-positive-ports-step* (*disc*,*sel*) *C m*) ⟹ *normalized-nnf-match m′*
apply(*simp add*: *normalize-positive-ports-step-def*)
apply(*elim exE conjE*, *rename-tac rst spts*)
apply(*drule sym*)
apply(*frule primitive-extractor-correct*(*2*)[*OF n wf-disc-sel*])

apply(*subgoal-tac getNeg spts* = []) **prefer** *2* **subgoal** **apply**(*drule primitive-extractor-correct*(*8*)[*OF n wf-disc-sel*]) **using** *noneg* **by** *simp+* **apply**(*simp split*: *match-compress.split-asm*) **by** *fastforce*

lemma *normalize-positive-ports-step-normalized-n-primitive*: **assumes** *n*: *normalized-nnf-match m* **and** *wf-disc-sel*: *wf-disc-sel* (*disc*,*sel*) *C* **and** *noneg*: ¬ *has-disc-negated disc False m* **shows** ∀ *m′* ∈ *set* (*normalize-positive-ports-step* (*disc*,*sel*) *C m*). *normalized-n-primitive* (*disc*,*sel*) (λ*ps. case ps of L4Ports - pts* ⇒ *length pts* ≤ *1*) *m′* **unfolding** *normalize-positive-ports-step-def* **apply**(*intro ballI*, *rename-tac m′*) **apply**(*simp*) **apply**(*elim exE conjE*, *rename-tac rst spts*) **apply**(*drule sym*) **apply**(*frule primitive-extractor-correct*(*2*)[*OF n wf-disc-sel*]) **apply**(*frule primitive-extractor-correct*(*3*)[*OF n wf-disc-sel*]) **apply**(*subgoal-tac getNeg spts* = []) **prefer** *2* **subgoal** **apply**(*drule primitive-extractor-correct*(*8*)[*OF n wf-disc-sel*]) **using** *noneg* **by** *simp+* **apply**(*subgoal-tac normalized-n-primitive* (*disc*,*sel*) (λ*ps. case ps of L4Ports - pts* ⇒ *length pts* ≤ *1*) *rst*) **prefer** *2* **subgoal** **by**(*drule*(*2*) *normalized-n-primitive-if-no-primitive*) **apply**(*simp split*: *match-compress.split-asm*) **using** *normalized-ports-singletonize-combine-rst*[*OF wf-disc-sel*] **by** *blast*

definition *normalize-positive-src-ports* :: *′i*::*len common-primitive match-expr* ⇒ *′i common-primitive match-expr list* **where** *normalize-positive-src-ports* = *normalize-positive-ports-step* (*is-Src-Ports*, *src-ports-sel*) *Src-Ports*

definition *normalize-positive-dst-ports* :: *′i*::*len common-primitive match-expr* ⇒ *′i common-primitive match-expr list* **where** *normalize-positive-dst-ports* = *normalize-positive-ports-step* (*is-Dst-Ports*, *dst-ports-sel*) *Dst-Ports*

lemma *noNeg-mapNegPos-helper*: *getNeg ls* = [] ⟹ *map* (*Pos* ∘ *C*) (*getPos ls*) = *NegPos-map C ls* **by**(*induction ls rule*: *getPos.induct*) *simp+*

lemma *normalize-positive-src-ports*: **assumes** *generic*: *primitive-matcher-generic* β **and** *n*: *normalized-nnf-match m* **and** *noneg*: ¬ *has-disc-negated is-Src-Ports False m*

shows
match-list (β, α) (*normalize-positive-src-ports m*) *a p* $\longleftrightarrow$ *matches* (β, α) *m a p*
apply(*simp add*: *normalize-positive-src-ports-def normalize-positive-ports-step-def*)
apply(*case-tac primitive-extractor* (*is-Src-Ports*, *src-ports-sel*) *m*, *rename-tac spts rst*)
apply(*simp*)
apply(*subgoal-tac getNeg spts* = [])
prefer *2* **subgoal**
apply(*drule primitive-extractor-correct*(*8*)[*OF n wf-disc-sel-common-primitive*(*1*)])
using *noneg* **by** *simp+*
apply(*simp*)
apply(*drule primitive-extractor-correct*(*1*)[*OF n wf-disc-sel-common-primitive*(*1*), **where** γ=(β, α) **and** *a*=*a* **and** *p*=*p*])
apply(*case-tac l4-ports-compress* (*getPos spts*))
apply(*simp*)
apply(*drule raw-ports-compress-src-CannotMatch*[*OF generic*, **where** α=α **and** *a*=*a* **and** *p*=*p*])
apply(*simp add*: *noNeg-mapNegPos-helper*; *fail*)
apply(*simp*)
apply(*drule raw-ports-compress-MatchesAll*[*OF generic*, **where** α=α **and** *a*=*a* **and** *p*=*p*])
apply(*simp add*: *noNeg-mapNegPos-helper*; *fail*)
apply(*simp add*: *bunch-of-lemmata-about-matches*)
apply(*drule raw-ports-compress-src-MatchExpr*[*OF generic*, **where** α=α **and** *a*=*a* **and** *p*=*p*])
apply(*insert singletonize-L4Ports-src*[*OF generic*, **where** α=α **and** *a*=*a* **and** *p*=*p*])
apply(*simp add*: *match-list-matches*)
apply(*simp add*: *bunch-of-lemmata-about-matches*)
apply(*simp add*: *noNeg-mapNegPos-helper*; *fail*)
done

lemma *normalize-positive-dst-ports*:
assumes *generic*: *primitive-matcher-generic* β
and *n*: *normalized-nnf-match m*
and *noneg*: $\neg$ *has-disc-negated is-Dst-Ports False m*
shows *match-list* (β, α) (*normalize-positive-dst-ports m*) *a p* $\longleftrightarrow$ *matches* (β, α) *m a p*
apply(*simp add*: *normalize-positive-dst-ports-def normalize-positive-ports-step-def*)
apply(*case-tac primitive-extractor* (*is-Dst-Ports*, *dst-ports-sel*) *m*, *rename-tac spts rst*)
apply(*simp*)
apply(*subgoal-tac getNeg spts* = [])
prefer *2* **subgoal**
apply(*drule primitive-extractor-correct*(*8*)[*OF n wf-disc-sel-common-primitive*(*2*)])
using *noneg* **by** *simp+*
apply(*simp*)

apply(*drule primitive-extractor-correct*(*1*)[*OF n wf-disc-sel-common-primitive*(*2*), **where** γ=(β, α) **and** *a*=*a* **and** *p*=*p*])
apply(*case-tac l4-ports-compress* (*getPos spts*))
apply(*simp*)
apply(*drule raw-ports-compress-dst-CannotMatch*[*OF generic*, **where** α=α **and** *a*=*a* **and** *p*=*p*])
apply(*simp add*: *noNeg-mapNegPos-helper*; *fail*)
apply(*simp*)
apply(*drule raw-ports-compress-MatchesAll*(*2*)[*OF generic*, **where** α=α **and** *a*=*a* **and** *p*=*p*])
apply(*simp add*: *noNeg-mapNegPos-helper*; *fail*)
apply(*simp add*: *bunch-of-lemmata-about-matches*)
apply(*drule raw-ports-compress-dst-MatchExpr*[*OF generic*, **where** α=α **and** *a*=*a* **and** *p*=*p*])
apply(*insert singletonize-L4Ports-dst*[*OF generic*, **where** α=α **and** *a*=*a* **and** *p*=*p*])
apply(*simp add*: *match-list-matches*)
apply(*simp add*: *bunch-of-lemmata-about-matches*)
apply(*simp add*: *noNeg-mapNegPos-helper*; *fail*)
done

lemma *normalize-positive-src-ports-nnf*:
assumes *n*: *normalized-nnf-match m*
and *noneg*: ¬ *has-disc-negated is-Src-Ports False m*
shows *m′* ∈ *set* (*normalize-positive-src-ports m*) ⟹ *normalized-nnf-match m′*
apply(*rule normalize-positive-ports-step-nnf*[*OF n wf-disc-sel-common-primitive*(*1*) *noneg*])
by(*simp add*: *normalize-positive-src-ports-def*)
lemma *normalize-positive-dst-ports-nnf*:
assumes *n*: *normalized-nnf-match m*
and *noneg*: ¬ *has-disc-negated is-Dst-Ports False m*
shows *m′* ∈ *set* (*normalize-positive-dst-ports m*) ⟹ *normalized-nnf-match m′*
apply(*rule normalize-positive-ports-step-nnf*[*OF n wf-disc-sel-common-primitive*(*2*) *noneg*])
by(*simp add*: *normalize-positive-dst-ports-def*)

lemma *normalize-positive-src-ports-normalized-n-primitive*:
assumes *n*: *normalized-nnf-match m*
and *noneg*: ¬ *has-disc-negated is-Src-Ports False m*
shows ∀ *m′* ∈ *set* (*normalize-positive-src-ports m*). *normalized-src-ports m′*
unfolding *normalized-src-ports-def2*
unfolding *normalize-positive-src-ports-def*
using *normalize-positive-ports-step-normalized-n-primitive*[*OF n wf-disc-sel-common-primitive*(*1*) *noneg*] **by** *blast*

lemma *normalize-positive-dst-ports-normalized-n-primitive*:
assumes *n*: *normalized-nnf-match m*
and *noneg*: ¬ *has-disc-negated is-Dst-Ports False m*

shows $\forall m' \in set\ (normalize\text{-}positive\text{-}dst\text{-}ports\ m).\ normalized\text{-}dst\text{-}ports\ m'$
unfolding *normalized-dst-ports-def2*
unfolding *normalize-positive-dst-ports-def*
using *normalize-positive-ports-step-normalized-n-primitive*[*OF n wf-disc-sel-common-primitive*(*2*) *noneg*] **by** *blast*

27.5 Complete Normalization

definition *normalize-ports-generic*
:: ('*i common-primitive match-expr* ⇒ '*i common-primitive match-expr list*) ⇒
('*i common-primitive match-expr* ⇒ '*i common-primitive match-expr*) ⇒
'*i*::*len common-primitive match-expr* ⇒ '*i common-primitive match-expr list*
where
normalize-ports-generic normalize-pos rewrite-neg m = *concat* (*map normalize-pos* (*normalize-match* (*rewrite-neg m*)))

lemma *normalize-ports-generic-nnf*:
assumes *n*: *normalized-nnf-match m*
and *inset*: $m' \in set\ (normalize\text{-}ports\text{-}generic\ normalize\text{-}pos\ rewrite\text{-}neg\ m)$
and *noNeg*: ¬ *has-disc-negated disc False* (*rewrite-neg m*)
and *normalize-nnf-pos*: $\bigwedge m\ m'.$
normalized-nnf-match m ⟹ ¬ *has-disc-negated disc False m* ⟹
$m' \in set\ (normalize\text{-}pos\ m) \Longrightarrow normalized\text{-}nnf\text{-}match\ m'$
shows *normalized-nnf-match m'*
using *inset* **apply**(*simp add*: *normalize-ports-generic-def*)
apply(*elim bexE*, *rename-tac a*)
apply(*subgoal-tac normalized-nnf-match a*)
prefer *2*
using *normalized-nnf-match-normalize-match* **apply** *blast*
apply(*erule normalize-nnf-pos*, *simp-all*)
apply(*rule not-has-disc-normalize-match*)
using *noNeg n* **by** *blast*+

lemma *normalize-ports-generic*:
assumes *n*: *normalized-nnf-match m*
and *normalize-pos*: $\bigwedge m.\ normalized\text{-}nnf\text{-}match\ m \Longrightarrow$ ¬ *has-disc-negated disc False m* ⟹
match-list γ (*normalize-pos m*) *a p* ⟷ *matches γ m a p*
and *rewrite-neg*: $\bigwedge m.\ normalized\text{-}nnf\text{-}match\ m \Longrightarrow$
matches γ (*rewrite-neg m*) *a p* = *matches γ m a p*
and *noNeg*: $\bigwedge m.\ normalized\text{-}nnf\text{-}match\ m \Longrightarrow$ ¬ *has-disc-negated disc False* (*rewrite-neg m*)
shows
match-list γ (*normalize-ports-generic normalize-pos rewrite-neg m*) *a p* ⟷ *matches γ m a p*
unfolding *normalize-ports-generic-def*
proof

have *1*: *ls* ∈ *set* (*normalize-match* (*rewrite-neg m*)) ⟹
match-list γ (*normalize-pos ls*) *a p* ⟹ *normalized-nnf-match ls* ⟹ *matches γ m a p*
for *ls*
apply(*subst*(*asm*) *normalize-pos*)
subgoal using *normalized-nnf-match-normalize-match* **by** *blast*
subgoal apply(*rule-tac m=rewrite-neg m* **in** *not-has-disc-normalize-match*)
using *noNeg n* **apply** *blast*
by *blast*
apply(*subgoal-tac matches γ* (*rewrite-neg m*) *a p*)
using *rewrite-neg*[*OF n*] **apply** *blast*
using *in-normalized-matches*[**where** *γ=γ* **and** *a=a* **and** *p=p*] **by** *blast*

show *match-list γ* (*concat* (*map normalize-pos* (*normalize-match* (*rewrite-neg m*)))) *a p* ⟹ *matches γ m a p*
apply(*simp add*: *match-list-concat*)
apply(*clarify*, *rename-tac ls*)
apply(*subgoal-tac normalized-nnf-match ls*)
using *1* **apply**(*simp*; *fail*)
using *normalized-nnf-match-normalize-match* **by** *blast*
next
have *1*: *ls* ∈ *set* (*normalize-match* (*rewrite-neg m*)) ⟹
matches γ ls a p ⟹
normalized-nnf-match ls ⟹
match-list γ (*concat* (*map normalize-pos* (*normalize-match* (*rewrite-neg m*)))) *a p* **for** *ls*
apply(*simp add*: *match-list-concat*)
apply(*rule-tac x=ls* **in** *bexI*)
prefer *2* **apply**(*simp*; *fail*)
apply(*subst normalize-pos*)
apply(*simp-all*)
apply(*rule-tac m=rewrite-neg m* **in** *not-has-disc-normalize-match*)
using *noNeg n* **apply** *blast*
by *blast*
show *matches γ m a p* ⟹ *match-list γ* (*concat* (*map normalize-pos* (*normalize-match* (*rewrite-neg m*)))) *a p*
apply(*subst*(*asm*) *rewrite-neg*[*OF n*, *symmetric*])
apply(*subst*(*asm*) *matches-to-match-list-normalize*)
apply(*subst*(*asm*) *match-list-matches*)
apply(*elim bexE*, *rename-tac ls*)
apply(*subgoal-tac normalized-nnf-match ls*)
using *1* **apply** *blast*
using *normalized-nnf-match-normalize-match* **by** *blast*
qed

lemma *normalize-ports-generic-normalized-n-primitive*:
assumes *n*: *normalized-nnf-match m* **and** *wf-disc-sel*: *wf-disc-sel* (*disc*,*sel*) *C*
and *noNeg*: ⋀*m*. *normalized-nnf-match m* ⟹ ¬ *has-disc-negated disc False*

```
(rewrite-neg m)
   and normalize-nnf-pos: ⋀m m'.
       normalized-nnf-match  m ⟹ ¬ has-disc-negated disc False m ⟹
        m' ∈ set (normalize-pos m) ⟹ normalized-nnf-match m'
   and normalize-pos: ⋀m m'.
       normalized-nnf-match m ⟹  ¬ has-disc-negated disc False m ⟹
        ∀ m'∈set (normalize-pos m).
              normalized-n-primitive (disc,sel) (λps. case ps of L4Ports - pts ⇒
length pts ≤ 1) m'
   shows ∀ m' ∈ set (normalize-ports-generic normalize-pos rewrite-neg m).
         normalized-n-primitive (disc,sel) (λps. case ps of L4Ports - pts ⇒ length
pts ≤ 1) m'
  unfolding normalize-ports-generic-def
  apply(intro ballI, rename-tac m')
  apply(simp)
  apply(elim bexE, rename-tac a)
  apply(subgoal-tac normalized-nnf-match a)
   prefer 2
   using normalized-nnf-match-normalize-match apply blast
  apply(subgoal-tac ¬ has-disc-negated disc False a)
   prefer 2
   subgoal for ls
    apply(rule-tac m=rewrite-neg m in not-has-disc-normalize-match)
     using noNeg n apply blast
    by blast
  apply(subgoal-tac normalized-nnf-match m')
   prefer 2
   using normalize-nnf-pos apply blast
  using normalize-pos by blast

  lemma normalize-ports-generic-normalize-positive-ports-step-erule:
    assumes n: normalized-nnf-match m
      and wf-disc-sel: wf-disc-sel (disc, sel) C
      and noProt: ∀ a. ¬ disc (Prot a)
      and P: P (disc2, sel2) m
      and P1: ⋀a. normalized-nnf-match a ⟹
                a ∈ set (normalize-match (rewrite-negated-primitives (disc, sel) C
l4-ports-negate-one m)) ⟹
             P (disc2, sel2) a
      and P2: ⋀a dpts rst. normalized-nnf-match a ⟹
                 primitive-extractor (disc, sel) a = (dpts, rst) ⟹
                 getNeg dpts = [] ⟹ P (disc2, sel2) a ⟹ P (disc2, sel2) rst
       and P3: ⋀ a spt rst. P (disc2, sel2) rst ⟹ P (disc2, sel2) (MatchAnd
(Match (C spt)) rst)
    shows m' ∈ set (normalize-ports-generic (normalize-positive-ports-step (disc,
sel) C) (rewrite-negated-primitives (disc, sel) C l4-ports-negate-one) m) ⟹
         P (disc2, sel2) m'
    using P apply(simp add: normalize-ports-generic-def)
    apply(elim bexE, rename-tac a)
```

apply(*subgoal-tac normalized-nnf-match a*)
prefer *2* **using** *normalized-nnf-match-normalize-match* **apply** *blast*
apply(*simp add*: *normalize-positive-ports-step-def*)
apply(*elim exE conjE*, *rename-tac rst dpts*)
apply(*drule sym*)
apply(*subgoal-tac getNeg dpts* = [])
prefer *2* **subgoal for** *a rst dpts*
apply(*erule iffD1*[*OF primitive-extractor-correct*(*8*)[*OF - wf-disc-sel*]])
apply(*simp*; *fail*)
apply(*rule not-has-disc-normalize-match*)
apply(*simp-all*)
apply(*rule rewrite-negated-primitives-not-has-disc-negated*[*OF n wf-disc-sel*])
apply(*intro allI*)
apply(*rule l4-ports-negate-one-not-has-disc-negated-generic*)
by(*simp add*: *noProt*)
apply(*subgoal-tac P* (*disc2*, *sel2*) *a*)
prefer *2* **subgoal**
apply(*rule P1*)
by(*simp*)
apply(*frule-tac a*=*a* **in** *P2*)
apply *blast*+
apply(*simp split*: *match-compress.split-asm*)
using *P3* **by** *auto*

lemma *normalize-ports-generic-preserves-normalized-n-primitive*:
assumes *n*: *normalized-nnf-match m*
and *wf-disc-sel*: *wf-disc-sel* (*disc*, *sel*) *C*
and *noProt*: ∀ *a*. ¬ *disc* (*Prot a*)
and *disc2-noC*: ∀ *a*. ¬ *disc2* (*C a*)
and *disc2-noProt*: (∀ *a*. ¬ *disc2* (*Prot a*)) ∨ ¬ *has-disc-negated disc False m*
shows *m*′ ∈ *set* (*normalize-ports-generic* (*normalize-positive-ports-step* (*disc*, *sel*) *C*) (*rewrite-negated-primitives* (*disc*, *sel*) *C l4-ports-negate-one*) *m*) ⟹
normalized-n-primitive (*disc2*, *sel2*) *f m* ⟹
normalized-n-primitive (*disc2*, *sel2*) *f m*′
thm *normalize-ports-generic-normalize-positive-ports-step-erule*
apply(*rule normalize-ports-generic-normalize-positive-ports-step-erule*[*OF n wf-disc-sel noProt*])
apply(*simp-all add*: *disc2-noC disc2-noProt*)
apply(*rule rewrite-negated-primitives-normalized-preserves-unrelated-helper*[*OF wf-disc-sel - - n*])
apply(*simp-all add*: *disc2-noC disc2-noProt*)
apply(*frule-tac m*=*a* **in** *primitive-extractor-correct*(*5*)[*OF - wf-disc-sel*, **where** *P*=*f*])
by *blast*+

lemma *normalize-ports-generic-preserves-normalized-not-has-disc*:
assumes *n*: *normalized-nnf-match m* **and** *nodisc*: ¬ *has-disc disc2 m*
and *wf-disc-sel*: *wf-disc-sel* (*disc*, *sel*) *C*

and *noProt*: $\forall$ *a*. $\neg$ *disc* (*Prot a*)
and *disc2-noC*: $\forall$ *a*. $\neg$ *disc2* (*C a*)
and *disc2-noProt*: ($\forall$ *a*. $\neg$ *disc2* (*Prot a*)) $\vee$ $\neg$ *has-disc-negated disc False m*
shows *m′*$\in$ *set* (*normalize-ports-generic* (*normalize-positive-ports-step* (*disc*, *sel*) *C*) (*rewrite-negated-primitives* (*disc*, *sel*) *C l4-ports-negate-one*) *m*)
$\Longrightarrow$ $\neg$ *has-disc disc2 m′*
apply(*rule normalize-ports-generic-normalize-positive-ports-step-erule*[*OF n wf-disc-sel noProt*])
apply(*simp-all add*: *disc2-noC disc2-noProt nodisc*)
subgoal for *a*
thm *normalize-match-preserves-nodisc*
apply(*rule-tac m*=*rewrite-negated-primitives* (*disc*, *sel*) *C l4-ports-negate-one m* **in** *normalize-match-preserves-nodisc*)
apply(*simp-all*)
apply(*insert disc2-noProt*)
apply(*elim disjE*)
thm *rewrite-negated-primitives-not-has-disc*[*of - disc2*]
subgoal apply(*rule rewrite-negated-primitives-not-has-disc*[*OF n wf-disc-sel nodisc - disc2-noC*])
using *l4-ports-negate-one-nodisc*[*OF disc2-noC*] **by** *blast*
using *rewrite-negated-primitives-preserves-not-has-disc*[*OF n wf-disc-sel nodisc - disc2-noC*] **by** *blast*
apply(*frule-tac m*=*a* **in** *primitive-extractor-correct*(*4*)[*OF - wf-disc-sel*])
by *blast*+

lemma *normalize-ports-generic-preserves-normalized-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m* **and** *nodisc*: $\neg$ *has-disc-negated disc2 False m*
and *wf-disc-sel*: *wf-disc-sel* (*disc*, *sel*) *C*
and *noProt*: $\forall$ *a*. $\neg$ *disc* (*Prot a*)
and *disc2-noProt*: ($\forall$ *a*. $\neg$ *disc2* (*Prot a*)) $\vee$ $\neg$ *has-disc-negated disc False m*
shows *m′*$\in$ *set* (*normalize-ports-generic* (*normalize-positive-ports-step* (*disc*, *sel*) *C*) (*rewrite-negated-primitives* (*disc*, *sel*) *C l4-ports-negate-one*) *m*)
$\Longrightarrow$ $\neg$ *has-disc-negated disc2 False m′*
apply(*rule normalize-ports-generic-normalize-positive-ports-step-erule*[*OF n wf-disc-sel noProt*])
apply(*simp-all add*: *disc2-noProt nodisc*)
subgoal for *a*
apply(*rule-tac m*=*rewrite-negated-primitives* (*disc*, *sel*) *C l4-ports-negate-one m* **in** *not-has-disc-normalize-match*)
apply(*simp-all*)
apply(*rule rewrite-negated-primitives-preserves-not-has-disc-negated*[*OF n wf-disc-sel*])
using *disc2-noProt l4-ports-negate-one-not-has-disc-negated-generic* **apply** *blast*
using *nodisc* **by** *blast*
subgoal for *a dpts rst*
apply(*frule-tac m*=*a* **and** *as*=*dpts* **and** *ms*=*rst* **and** *neg*=*False* **in** *primitive-extractor-correct*(*6*)[*OF - wf-disc-sel*])

by *blast+*
done

definition *normalize-src-ports*
:: *'i*::*len common-primitive match-expr* ⇒ *'i common-primitive match-expr list*
where
normalize-src-ports m = *normalize-ports-generic normalize-positive-src-ports rewrite-negated-src-ports m*

definition *normalize-dst-ports*
:: *'i*::*len common-primitive match-expr* ⇒ *'i common-primitive match-expr list*
where
normalize-dst-ports m = *normalize-ports-generic normalize-positive-dst-ports rewrite-negated-dst-ports m*

lemma *normalize-src-ports*:
assumes *generic*: *primitive-matcher-generic* β
and *n*: *normalized-nnf-match m*
shows *match-list* (β, α) (*normalize-src-ports m*) *a p* ⟷ *matches* (β, α) *m a p*
apply(*simp add*: *normalize-src-ports-def*)
apply(*rule normalize-ports-generic*[*OF n*])
using *normalize-positive-src-ports*[*OF generic*]
rewrite-negated-src-ports[*OF generic*, **where** α=α **and** *a*=*a* **and** *p*=*p*]
rewrite-negated-src-ports-not-has-disc-negated **by** *blast+*

lemma *normalize-dst-ports*:
assumes *generic*: *primitive-matcher-generic* β
and *n*: *normalized-nnf-match m*
shows *match-list* (β, α) (*normalize-dst-ports m*) *a p* ⟷ *matches* (β, α) *m a p*
apply(*simp add*: *normalize-dst-ports-def*)
apply(*rule normalize-ports-generic*[*OF n*])
using *normalize-positive-dst-ports*[*OF generic*]
rewrite-negated-dst-ports[*OF generic*, **where** α=α **and** *a*=*a* **and** *p*=*p*]
rewrite-negated-dst-ports-not-has-disc-negated **by** *blast+*

lemma *normalize-src-ports-normalized-n-primitive*:
assumes *n*:*normalized-nnf-match m*
shows ∀ *m'* ∈ *set* (*normalize-src-ports m*). *normalized-src-ports m'*
unfolding *normalize-src-ports-def normalized-src-ports-def2*
apply(*rule normalize-ports-generic-normalized-n-primitive*[*OF n wf-disc-sel-common-primitive*(*1*)])
using *rewrite-negated-src-ports-not-has-disc-negated* **apply** *blast*
using *normalize-positive-src-ports-nnf* **apply** *blast*
unfolding *normalized-src-ports-def2*[*symmetric*]
using *normalize-positive-src-ports-normalized-n-primitive* **by** *blast*

lemma *normalize-dst-ports-normalized-n-primitive*:
assumes *n*: *normalized-nnf-match m*
shows ∀ *m'* ∈ *set* (*normalize-dst-ports m*). *normalized-dst-ports m'*
unfolding *normalize-dst-ports-def normalized-dst-ports-def2*

apply(*rule normalize-ports-generic-normalized-n-primitive*[*OF n wf-disc-sel-common-primitive*(*2*)])
using *rewrite-negated-dst-ports-not-has-disc-negated* **apply** *blast*
using *normalize-positive-dst-ports-nnf* **apply** *blast*
unfolding *normalized-dst-ports-def2*[*symmetric*]
using *normalize-positive-dst-ports-normalized-n-primitive* **by** *blast*

lemma *normalize-src-ports-nnf*:
assumes *n*: *normalized-nnf-match m*
shows $m' \in$ *set* (*normalize-src-ports m*) $\Longrightarrow$ *normalized-nnf-match* m'
apply(*simp add*: *normalize-src-ports-def*)
apply(*erule normalize-ports-generic-nnf*[*OF n*])
using *n rewrite-negated-src-ports-not-has-disc-negated* **apply** *blast*
using *normalize-positive-src-ports-nnf* **by** *blast*

lemma *normalize-dst-ports-nnf*:
assumes *n*: *normalized-nnf-match m*
shows $m' \in$ *set* (*normalize-dst-ports m*) $\Longrightarrow$ *normalized-nnf-match* m'
apply(*simp add*: *normalize-dst-ports-def*)
apply(*erule normalize-ports-generic-nnf*[*OF n*])
using *n rewrite-negated-dst-ports-not-has-disc-negated* **apply** *blast*
using *normalize-positive-dst-ports-nnf* **by** *blast*

lemma *normalize-src-ports-preserves-normalized-n-primitive*:
assumes *n*: *normalized-nnf-match m*
and *disc2-noC*: $\forall a. \neg$ *disc2* (*Src-Ports a*)
and *disc2-noProt*: $(\forall a. \neg$ *disc2* (*Prot a*)$) \vee \neg$ *has-disc-negated is-Src-Ports False m*
shows $m' \in$ *set* (*normalize-src-ports m*) $\Longrightarrow$
normalized-n-primitive (*disc2*, *sel2*) *f m* $\Longrightarrow$
normalized-n-primitive (*disc2*, *sel2*) *f* m'
apply(*rule normalize-ports-generic-preserves-normalized-n-primitive*[*OF n wf-disc-sel-common-primitive*(*1*)])
by(*simp-all add*: *disc2-noC disc2-noProt normalize-src-ports-def normalize-ports-generic-def*
normalize-positive-src-ports-def rewrite-negated-src-ports-def)

lemma *normalize-dst-ports-preserves-normalized-n-primitive*:
assumes *n*: *normalized-nnf-match m*
and *disc2-noC*: $\forall a. \neg$ *disc2* (*Dst-Ports a*)
and *disc2-noProt*: $(\forall a. \neg$ *disc2* (*Prot a*)$) \vee \neg$ *has-disc-negated is-Dst-Ports False m*
shows $m' \in$ *set* (*normalize-dst-ports m*) $\Longrightarrow$
normalized-n-primitive (*disc2*, *sel2*) *f m* $\Longrightarrow$
normalized-n-primitive (*disc2*, *sel2*) *f* m'
apply(*rule normalize-ports-generic-preserves-normalized-n-primitive*[*OF n wf-disc-sel-common-primitive*(*2*)])
by(*simp-all add*: *disc2-noC disc2-noProt normalize-dst-ports-def normalize-ports-generic-def*

normalize-positive-dst-ports-def rewrite-negated-dst-ports-def)

lemma *normalize-src-ports-preserves-normalized-not-has-disc*:
assumes *n*: *normalized-nnf-match m* **and** *nodisc*: ¬ *has-disc disc2 m*
and *disc2-noC*: ∀ *a*. ¬ *disc2* (*Src-Ports a*)
and *disc2-noProt*: (∀ *a*. ¬ *disc2* (*Prot a*)) ∨ ¬ *has-disc-negated is-Src-Ports False m*
shows *m'*∈ *set* (*normalize-src-ports m*)
⟹ ¬ *has-disc disc2 m'*
apply(*rule normalize-ports-generic-preserves-normalized-not-has-disc*[*OF n nodisc wf-disc-sel-common-primitive*(*1*)])
apply(*simp add*: *disc2-noC disc2-noProt*)+
by (*simp add*: *normalize-ports-generic-def normalize-positive-src-ports-def normalize-src-ports-def rewrite-negated-src-ports-def*)

lemma *normalize-dst-ports-preserves-normalized-not-has-disc*:
assumes *n*: *normalized-nnf-match m* **and** *nodisc*: ¬ *has-disc disc2 m*
and *disc2-noC*: ∀ *a*. ¬ *disc2* (*Dst-Ports a*)
and *disc2-noProt*: (∀ *a*. ¬ *disc2* (*Prot a*)) ∨ ¬ *has-disc-negated is-Dst-Ports False m*
shows *m'*∈ *set* (*normalize-dst-ports m*)
⟹ ¬ *has-disc disc2 m'*
apply(*rule normalize-ports-generic-preserves-normalized-not-has-disc*[*OF n nodisc wf-disc-sel-common-primitive*(*2*)])
apply(*simp add*: *disc2-noC disc2-noProt*)+
by (*simp add*: *normalize-ports-generic-def normalize-positive-dst-ports-def normalize-dst-ports-def rewrite-negated-dst-ports-def*)

lemma *normalize-src-ports-preserves-normalized-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m* **and** *nodisc*: ¬ *has-disc-negated disc2 False m*
and *disc2-noProt*: (∀ *a*. ¬ *disc2* (*Prot a*)) ∨ ¬ *has-disc-negated is-Src-Ports False m*
shows *m'*∈ *set* (*normalize-src-ports m*)
⟹ ¬ *has-disc-negated disc2 False m'*
apply(*rule normalize-ports-generic-preserves-normalized-not-has-disc-negated*[*OF n nodisc wf-disc-sel-common-primitive*(*1*)])
apply(*simp add*: *disc2-noProt*)+
by (*simp add*: *normalize-ports-generic-def normalize-positive-src-ports-def normalize-src-ports-def rewrite-negated-src-ports-def*)

lemma *normalize-dst-ports-preserves-normalized-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m* **and** *nodisc*: ¬ *has-disc-negated disc2 False m*
and *disc2-noProt*: (∀ *a*. ¬ *disc2* (*Prot a*)) ∨ ¬ *has-disc-negated is-Dst-Ports False m*
shows *m'*∈ *set* (*normalize-dst-ports m*)
⟹ ¬ *has-disc-negated disc2 False m'*
apply(*rule normalize-ports-generic-preserves-normalized-not-has-disc-negated*[*OF*

n nodisc wf-disc-sel-common-primitive(2)])
apply(*simp add*: *disc2-noProt*)+
by (*simp add*: *normalize-ports-generic-def normalize-positive-dst-ports-def normalize-dst-ports-def rewrite-negated-dst-ports-def*)

value[*code*] *normalize-src-ports*
(*MatchAnd* (*Match* (*Dst* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*127*, *0*, *0*, *0*)) *8*)))
(*MatchAnd* (*Match* (*Prot* (*Proto TCP*)))
(*MatchNot* (*Match* (*Src-Ports* (*L4Ports UDP* [(*80*,*80*)])))))
))

lemma *map opt-MatchAny-match-expr* (*normalize-src-ports*
(*MatchAnd* (*Match* (*Dst* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*127*, *0*, *0*, *0*)) *8*)))
(*MatchAnd* (*Match* (*Prot* (*Proto TCP*)))
(*MatchNot* (*Match* (*Src-Ports* (*L4Ports UDP* [(*80*,*80*)])))))
))) =
[*MatchAnd* (*MatchNot* (*Match* (*Prot* (*Proto UDP*)))) (*MatchAnd* (*Match* (*Dst* (*IpAddrNetmask 0x7F000000 8*))) (*Match* (*Prot* (*Proto TCP*)))),
MatchAnd (*Match* (*Src-Ports* (*L4Ports UDP* [(*0*, *79*)]))) (*MatchAnd* (*Match* (*Dst* (*IpAddrNetmask 0x7F000000 8*))) (*Match* (*Prot* (*Proto TCP*)))),
MatchAnd (*Match* (*Src-Ports* (*L4Ports UDP* [(*81*, *0xFFFF*)]))) (*MatchAnd* (*Match* (*Dst* (*IpAddrNetmask 0x7F000000 8*))) (*Match* (*Prot* (*Proto TCP*))))] **by** *eval*

lemma *map opt-MatchAny-match-expr* (*normalize-src-ports*
(*MatchAnd* (*Match* (*Dst* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*127*, *0*, *0*, *0*)) *8*)))
(*MatchAnd* (*Match* (*Prot* (*Proto ICMP*)))
(*MatchAnd* (*Match* (*Src-Ports* (*L4Ports TCP* [(*22*,*22*)])))
(*MatchNot* (*Match* (*Src-Ports* (*L4Ports UDP* [(*80*,*80*)])))))
))))
=
[*MatchAnd* (*Match* (*Src-Ports* (*L4Ports TCP* [(*22*, *22*)])))
(*MatchAnd* (*MatchNot* (*Match* (*Prot* (*Proto UDP*)))) (*MatchAnd* (*Match* (*Dst* (*IpAddrNetmask 0x7F000000 8*))) (*Match* (*Prot* (*Proto ICMP*)))))] **by** *eval*

lemma *map opt-MatchAny-match-expr* (*normalize-src-ports*
(*MatchAnd* (*Match* ((*Src-Ports* (*L4Ports UDP* [(*21*,*21*), (*22*,*22*)])) :: *32 common-primitive*))
(*Match* (*Prot* (*Proto UDP*)))))
=
[*MatchAnd* (*Match* (*Src-Ports* (*L4Ports UDP* [(*21*, *22*)]))) (*Match* (*Prot* (*Proto UDP*)))] **by** *eval*

lemma *normalize-match* (*andfold-MatchExp* (*map* (*l4-ports-negate-one C*) [])) =

[*MatchAny*] **by**(*simp*)

definition *replace-primitive-matchexpr*
:: ((*'a* ⇒ *bool*) × (*'a* ⇒ *'b*)) ⇒ — *disc-sel*
(*'b negation-type* ⇒ *'a match-expr*) ⇒ — replace function
'a match-expr ⇒ *'a match-expr* **where**
replace-primitive-matchexpr disc-sel replace-f m ≡
let (*as*, *rst*) = *primitive-extractor disc-sel m*
in if as = [] *then m else*
MatchAnd
(*andfold-MatchExp* (*map replace-f as*))
rst

It does nothing of there is not even a primitive in it

lemma *replace-primitive-matchexpr-unchanged-if-not-has-disc*:
assumes *n*: *normalized-nnf-match m*
and *wf-disc-sel*: *wf-disc-sel* (*disc*,*sel*) *C*
and *noDisc*: ¬ *has-disc disc m*
shows *replace-primitive-matchexpr* (*disc*,*sel*) *replace-f m* = *m*
apply(*simp add*: *replace-primitive-matchexpr-def*)
apply(*case-tac primitive-extractor* (*disc*,*sel*) *m*, *rename-tac spts rst*)
apply(*simp*)
apply(*frule primitive-extractor-correct*(7)[*OF n wf-disc-sel*])
using *noDisc* **by** *blast*+

lemma *replace-primitive-matchexpr*:
assumes *n*: *normalized-nnf-match m* **and** *wf-disc-sel*: *wf-disc-sel disc-sel C*
and *replace-f*: ∀ *pt*. *matches* γ (*replace-f pt*) *a p* ⟷
matches γ (*negation-type-to-match-expr-f C pt*) *a p*
shows *matches* γ (*replace-primitive-matchexpr disc-sel replace-f m*) *a p* ⟷ *matches* γ *m a p*
proof −
obtain *spts rst* **where** *pext*: *primitive-extractor disc-sel m* = (*spts*, *rst*)
by(*cases primitive-extractor disc-sel m*) *simp*
obtain *disc sel* **where** *disc-sel*: *disc-sel* = (*disc*, *sel*) **by**(*cases disc-sel*) *simp*
with *wf-disc-sel* **have** *wf-disc-sel'*: *wf-disc-sel* (*disc*, *sel*) *C* **by** *simp*
from *disc-sel pext* **have** *pext'*: *primitive-extractor* (*disc*, *sel*) *m* = (*spts*, *rst*) **by** *simp*

have *matches* γ (*andfold-MatchExp* (*map replace-f spts*)) *a p* ∧ *matches* γ *rst a p* ⟷
matches γ *m a p*

apply(*subst primitive-extractor-correct*(*1*)[*OF n wf-disc-sel' pext'*, *symmetric*])
apply(*simp add*: *andfold-MatchExp-matches*)
apply(*simp add*: *replace-f*)
using *alist-and-negation-type-to-match-expr-f-matches* **by** *fast*
thus *?thesis* **by**(*simp add*: *replace-primitive-matchexpr-def pext bunch-of-lemmata-about-matches*)
qed

lemma *replace-primitive-matchexpr-replaces-disc*:
assumes *n*: *normalized-nnf-match m* **and** *wf-disc-sel*: *wf-disc-sel* (*disc*, *sel*) *C*
and *replace-f*: ∀ *a*. ¬ *has-disc disc* (*replace-f a*)
shows ¬ *has-disc disc* (*replace-primitive-matchexpr* (*disc*, *sel*) *replace-f m*)
apply(*simp add*: *replace-primitive-matchexpr-def*)
apply(*case-tac primitive-extractor* (*disc*,*sel*) *m*, *rename-tac spts rst*)
apply(*simp*)
apply(*frule primitive-extractor-correct*(*3*)[*OF n wf-disc-sel*])
apply *simp*
apply(*frule primitive-extractor-correct*(*7*)[*OF n wf-disc-sel*])
apply *simp*
apply(*case-tac* ¬ *has-disc disc m*)
apply(*simp*)
apply(*simp*)
apply(*frule*(*1*) *primitive-extractor-correct*(*9*)[*OF n wf-disc-sel*])
apply(*simp*)
apply(*rule MatchExpr-Fold.andfold-MatchExp-not-discI*)
using *replace-f* **by** *simp*

lemma *replace-primitive-matchexpr-preserves-not-has-disc*:
assumes *n*: *normalized-nnf-match m* **and** *wf-disc-sel*: *wf-disc-sel* (*disc*,*sel*) *C*
and *nodisc*: ¬ *has-disc disc2 m*
and *replace-f*: *has-disc disc m* ⟹ ∀ *pts*. ¬ *has-disc disc2* (*replace-f pts*)
shows ¬ *has-disc disc2* (*replace-primitive-matchexpr* (*disc*,*sel*) *replace-f m*)
apply(*simp add*: *replace-primitive-matchexpr-def*)
apply(*case-tac primitive-extractor* (*disc*,*sel*) *m*, *rename-tac spts rst*)
apply(*simp*)
apply(*frule primitive-extractor-correct*(*4*)[*OF n wf-disc-sel*])
apply(*case-tac* ¬ *has-disc disc m*)
subgoal
apply(*frule primitive-extractor-correct*(*7*)[*OF n wf-disc-sel*])
using *nodisc* **by** *blast*
apply(*simp*)
apply(*intro conjI impI*)
using *nodisc* **apply**(*simp*; *fail*)
apply(*rule andfold-MatchExp-not-discI*)
apply(*simp add*: *replace-f*; *fail*)
using *nodisc* **by** *blast*

lemma *normalize-replace-primitive-matchexpr-preserves-normalized-n-primitive*:
assumes *n*: *normalized-nnf-match m*

and *wf-disc-sel*: *wf-disc-sel* (*disc*, *sel*) *C*
and *replace-f*:
⋀*a m'*. *m'* ∈ *set* (*normalize-match* (*replace-f a*)) ⟹ *normalized-n-primitive* (*disc2*, *sel2*) *f m'*
and *nprim*: *normalized-n-primitive* (*disc2*, *sel2*) *f m*
and *m'*: *m'* ∈ *set* (*normalize-match* (*replace-primitive-matchexpr* (*disc*,*sel*) *replace-f m*))
shows *normalized-n-primitive* (*disc2*, *sel2*) *f m'*
proof −
have *x*: *x* ∈ *set* (*normalize-match* (*andfold-MatchExp* (*map replace-f as*))) ⟹ *normalized-n-primitive* (*disc2*, *sel2*) *f x* **for** *x as*
apply(*rule normalize-andfold-MatchExp-normalized-n-primitive*)
apply(*simp-all*)
using *replace-f* **by** *blast*
from *m'* **show** *?thesis*
apply(*simp add*: *replace-primitive-matchexpr-def*)
apply(*case-tac primitive-extractor* (*disc*, *sel*) *m*, *rename-tac as rst*)
apply(*simp split*: *if-split-asm*)
using *normalize-match-preserves-normalized-n-primitive nprim* **apply** *blast*
apply(*frule-tac P=f* **in** *primitive-extractor-correct*(*5*)[*OF n wf-disc-sel*])
apply(*clarify*)
apply(*simp*)
apply(*intro conjI*)
prefer *2*
using *normalize-match-preserves-normalized-n-primitive nprim* **apply** *blast*
by(*simp add*: *x*)
qed

lemma *normalize-replace-primitive-matchexpr-preserves-normalized-not-has-disc*:
assumes *n*: *normalized-nnf-match m*
and *wf-disc-sel*: *wf-disc-sel* (*disc*, *sel*) *C*
and *nodisc*: ¬ *has-disc disc2 m*
and *replace-f*: ⋀*a*. ¬ *has-disc disc2* (*replace-f a*)
shows *m'*∈ *set* (*normalize-match* (*replace-primitive-matchexpr* (*disc*,*sel*) *replace-f m*))
⟹ ¬ *has-disc disc2 m'*
apply(*simp add*: *replace-primitive-matchexpr-def*)
apply(*case-tac primitive-extractor* (*disc*, *sel*) *m*, *rename-tac as rst*)
apply(*simp split*: *if-split-asm*)
using *nodisc normalize-match-preserves-nodisc* **apply** *blast*
apply(*frule primitive-extractor-correct*(*4*)[*OF n wf-disc-sel*])
apply(*elim bexE*, *rename-tac x*)
apply(*erule Set.imageE*, *rename-tac xright*)
apply(*simp*)
apply(*intro conjI*)
apply(*rule normalize-match-preserves-nodisc*, *simp-all*)
apply(*rule andfold-MatchExp-not-discI*, *simp*)
using *replace-f* **apply** *blast*
apply(*rule normalize-match-preserves-nodisc*)

apply(*insert nodisc*)
by(*simp-all*)

lemma *normalize-replace-primitive-matchexpr-preserves-normalized-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m*
and *wf-disc-sel*: *wf-disc-sel* (*disc*, *sel*) *C*
and *nodisc*: ¬ *has-disc-negated disc2 neg m*
and *replace-f*: ⋀*a*. ¬ *has-disc-negated disc2 neg* (*replace-f a*)
shows *m′*∈ *set* (*normalize-match* (*replace-primitive-matchexpr* (*disc*,*sel*) *replace-f m*))
⟹ ¬ *has-disc-negated disc2 neg m′*
apply(*simp add*: *replace-primitive-matchexpr-def*)
apply(*case-tac primitive-extractor* (*disc*, *sel*) *m*, *rename-tac as rst*)
apply(*simp split*: *if-split-asm*)
using *nodisc not-has-disc-normalize-match* **apply** *blast*
apply(*frule primitive-extractor-correct*(*6*)[*OF n wf-disc-sel*, **where** *neg*=*neg*])
apply(*elim bexE*, *rename-tac x*)
apply(*erule Set.imageE*, *rename-tac xright*)
apply(*simp*)
apply(*intro conjI*)
apply(*rule not-has-disc-normalize-match*, *simp-all*)
apply(*rule andfold-MatchExp-not-disc-negatedI*, *simp*)
using *replace-f* **apply** *blast*
apply(*rule not-has-disc-normalize-match*)
apply(*insert nodisc*)
by(*simp-all*)

corollary *normalize-replace-primitive-matchexpr*:
assumes *n*: *normalized-nnf-match m*
and *replace-f*:
⋀*m*. *normalized-nnf-match m* ⟹
matches γ (*replace-primitive-matchexpr disc-sel replace-f m*) *a p* ⟷ *matches γ m a p*
shows
match-list γ (*normalize-match* (*replace-primitive-matchexpr disc-sel replace-f m*)) *a p* ⟷
matches γ m a p
by(*simp add*: *matches-to-match-list-normalize*[*symmetric*] *replace-f n*)

fun *rewrite-MultiportPorts-one*
:: *ipt-l4-ports negation-type*⇒ *'i*::*len common-primitive match-expr* **where**
rewrite-MultiportPorts-one (*Pos pts*) =
MatchOr (*Match* (*Src-Ports pts*)) (*Match* (*Dst-Ports pts*)) |
rewrite-MultiportPorts-one (*Neg pts*) =
MatchAnd (*MatchNot* (*Match* (*Src-Ports pts*))) (*MatchNot* (*Match* (*Dst-Ports pts*)))

lemma *rewrite-MultiportPorts-one*:

assumes *generic*: *primitive-matcher-generic* β **and** *n*: *normalized-nnf-match m*
shows
matches (β, α) (*replace-primitive-matchexpr* (*is-MultiportPorts*, *multiportports-sel*) *rewrite-MultiportPorts-one m*) *a p* $\longleftrightarrow$
matches (β, α) *m a p*
apply(*rule replace-primitive-matchexpr*[*OF n wf-disc-sel-common-primitive*(*11*)])
apply(*rule allI*, *rename-tac pt*)
apply(*case-tac pt*)
apply(*simp add*: *primitive-matcher-generic.MultiportPorts-single-rewrite-MatchOr*[*OF generic*]; *fail*)
apply(*simp add*: *primitive-matcher-generic.MultiportPorts-single-not-rewrite-MatchAnd*[*OF generic*]; *fail*)
done

lemma $\forall$ *a*. $\neg$ *disc* (*Src-Ports a*) $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc* (*Dst-Ports a*) $\Longrightarrow$
normalized-n-primitive (*disc*, *sel*) *f m* $\Longrightarrow$
$\forall$ *m*$'$ $\in$ *set* (*normalize-match* (*rewrite-MultiportPorts-one a*)).
normalized-n-primitive (*disc*, *sel*) *f m*$'$
apply(*cases a*)
by(*simp-all add*: *MatchOr-def*)

lemma *rewrite-MultiportPorts-one-nodisc*:
$\forall$ *a*. $\neg$ *disc* (*Src-Ports a*) $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc* (*Dst-Ports a*) $\Longrightarrow$
$\neg$ *has-disc disc* (*rewrite-MultiportPorts-one a*)
$\forall$ *a*. $\neg$ *disc* (*Src-Ports a*) $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc* (*Dst-Ports a*) $\Longrightarrow$
$\neg$ *has-disc-negated disc neg* (*rewrite-MultiportPorts-one a*)
by(*cases a*, *simp-all add*: *MatchOr-def*)+

definition *rewrite-MultiportPorts*
:: *'i*::*len common-primitive match-expr* $\Rightarrow$ *'i common-primitive match-expr list*
where
rewrite-MultiportPorts m $\equiv$ *normalize-match*
(*replace-primitive-matchexpr* (*is-MultiportPorts*, *multiportports-sel*) *rewrite-MultiportPorts-one m*)

lemma *rewrite-MultiportPorts*:
assumes *generic*: *primitive-matcher-generic* β
and *n*: *normalized-nnf-match m*
shows
match-list (β, α) (*rewrite-MultiportPorts m*) *a p* $\longleftrightarrow$ *matches* (β, α) *m a p*
unfolding *rewrite-MultiportPorts-def*
apply(*intro normalize-replace-primitive-matchexpr*[*OF n*])
by(*simp add*: *rewrite-MultiportPorts-one*[*OF generic*])

lemma *rewrite-MultiportPorts-normalized-nnf-match*:
m$'$ $\in$ *set* (*rewrite-MultiportPorts m*) $\Longrightarrow$ *normalized-nnf-match m*$'$
apply(*simp add*: *rewrite-MultiportPorts-def*)
using *normalized-nnf-match-normalize-match* **by** *blast*

It does nothing of there is not even the primitive in it

lemma *rewrite-MultiportPorts-unchanged-if-not-has-disc*:
assumes *n*: *normalized-nnf-match m*
and *noDisc*: ¬ *has-disc is-MultiportPorts m*
shows *rewrite-MultiportPorts m* = [*m*]
apply(*simp add*: *rewrite-MultiportPorts-def*)
apply(*subst replace-primitive-matchexpr-unchanged-if-not-has-disc*[*OF n*
wf-disc-sel-common-primitive(*11*) *noDisc*])
using *n* **by**(*fact normalize-match-already-normalized*)

lemma *rewrite-MultiportPorts-preserves-normalized-n-primitive*:
assumes *n*: *normalized-nnf-match m*
and *disc2-noSrcPorts*: ∀ *a*. ¬ *disc2* (*Src-Ports a*)
and *disc2-noDstPorts*: ∀ *a*. ¬ *disc2* (*Dst-Ports a*)
shows *m′* ∈ *set* (*rewrite-MultiportPorts m*) ⟹
normalized-n-primitive (*disc2*, *sel2*) *f m* ⟹
normalized-n-primitive (*disc2*, *sel2*) *f m′*
unfolding *rewrite-MultiportPorts-def*
apply(*rule normalize-replace-primitive-matchexpr-preserves-normalized-n-primitive*[*OF*
n wf-disc-sel-common-primitive(*11*)])
apply *simp-all*
apply(*rename-tac a a′*)
apply(*case-tac a*)
apply(*simp-all add*: *MatchOr-def*)
using *disc2-noSrcPorts disc2-noDstPorts* **by** *fastforce*+

lemma *rewrite-MultiportPorts-preserves-normalized-not-has-disc*:
assumes *n*: *normalized-nnf-match m*
and *nodisc*: ¬ *has-disc disc2 m*
and *disc2-noSrcPorts*: ∀ *a*. ¬ *disc2* (*Src-Ports a*)
and *disc2-noDstPorts*: ∀ *a*. ¬ *disc2* (*Dst-Ports a*)
shows *m′*∈ *set* (*rewrite-MultiportPorts m*)
⟹ ¬ *has-disc disc2 m′*
apply(*simp add*: *rewrite-MultiportPorts-def*)
apply(*rule normalize-replace-primitive-matchexpr-preserves-normalized-not-has-disc*[*OF*
n wf-disc-sel-common-primitive(*11*) *nodisc*])
by(*simp-all add*: *rewrite-MultiportPorts-one-nodisc disc2-noSrcPorts disc2-noDstPorts*)

lemma *rewrite-MultiportPorts-preserves-normalized-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m*
and *nodisc*: ¬ *has-disc-negated disc2 neg m*
and *disc2-noSrcPorts*: ∀ *a*. ¬ *disc2* (*Src-Ports a*)
and *disc2-noDstPorts*: ∀ *a*. ¬ *disc2* (*Dst-Ports a*)
shows *m′*∈ *set* (*rewrite-MultiportPorts m*)
⟹ ¬ *has-disc-negated disc2 neg m′*
apply(*simp add*: *rewrite-MultiportPorts-def*)
apply(*rule normalize-replace-primitive-matchexpr-preserves-normalized-not-has-disc-negated*[*OF*

n wf-disc-sel-common-primitive(*11*) *nodisc*])
by(*simp-all add*: *rewrite-MultiportPorts-one-nodisc disc2-noSrcPorts disc2-noDstPorts*)

lemma *rewrite-MultiportPorts-removes-MultiportsPorts*:
assumes *n*: *normalized-nnf-match m*
shows $m' \in$ *set* (*rewrite-MultiportPorts m*) $\Longrightarrow \neg$ *has-disc is-MultiportPorts* m'
apply(*simp add*: *rewrite-MultiportPorts-def*)
apply(*rule normalize-match-preserves-nodisc*)
apply(*simp-all*)
apply(*rule replace-primitive-matchexpr-replaces-disc*[*OF n wf-disc-sel-common-primitive*(*11*)])
apply(*intro allI*, *rename-tac a*)
by(*case-tac a*, *simp-all add*: *MatchOr-def*)

end
theory *IpAddresses-Normalize*
imports *Common-Primitive-Lemmas*
begin

27.6 Normalizing IP Addresses

fun *normalized-src-ips* :: *'i*::*len common-primitive match-expr* ⇒ *bool* **where**
normalized-src-ips MatchAny = *True* |
normalized-src-ips (*Match* (*Src* (*IpAddrRange* - -))) = *False* |
normalized-src-ips (*Match* (*Src* (*IpAddr* -))) = *False* |
normalized-src-ips (*Match* (*Src* (*IpAddrNetmask* - -))) = *True* |
normalized-src-ips (*Match* -) = *True* |
normalized-src-ips (*MatchNot* (*Match* (*Src* -))) = *False* |
normalized-src-ips (*MatchNot* (*Match* -)) = *True* |
normalized-src-ips (*MatchAnd m1 m2*) = (*normalized-src-ips m1* ∧ *normalized-src-ips m2*) |
normalized-src-ips (*MatchNot* (*MatchAnd* - -)) = *False* |
normalized-src-ips (*MatchNot* (*MatchNot* -)) = *False* |
normalized-src-ips (*MatchNot* (*MatchAny*)) = *True*

lemma *normalized-src-ips-def2*: *normalized-src-ips ms* = *normalized-n-primitive* (*is-Src*, *src-sel*) *normalized-cidr-ip ms*
by(*induction ms rule*: *normalized-src-ips.induct*, *simp-all add*: *normalized-cidr-ip-def*)

fun *normalized-dst-ips* :: *'i*::*len common-primitive match-expr* ⇒ *bool* **where**
normalized-dst-ips MatchAny = *True* |
normalized-dst-ips (*Match* (*Dst* (*IpAddrRange* - -))) = *False* |
normalized-dst-ips (*Match* (*Dst* (*IpAddr* -))) = *False* |
normalized-dst-ips (*Match* (*Dst* (*IpAddrNetmask* - -))) = *True* |
normalized-dst-ips (*Match* -) = *True* |
normalized-dst-ips (*MatchNot* (*Match* (*Dst* -))) = *False* |
normalized-dst-ips (*MatchNot* (*Match* -)) = *True* |
normalized-dst-ips (*MatchAnd m1 m2*) = (*normalized-dst-ips m1* ∧ *normalized-dst-ips m2*) |

normalized-dst-ips (*MatchNot* (*MatchAnd* - -)) = *False* |
normalized-dst-ips (*MatchNot* (*MatchNot* -)) = *False* |
normalized-dst-ips (*MatchNot MatchAny*) = *True*

lemma *normalized-dst-ips-def2*: *normalized-dst-ips ms* = *normalized-n-primitive* (*is-Dst*, *dst-sel*) *normalized-cidr-ip ms*
by(*induction ms rule*: *normalized-dst-ips.induct*, *simp-all add*: *normalized-cidr-ip-def*)

value *normalize-primitive-extract* (*is-Src*, *src-sel*) *Src ipt-iprange-compress* (*MatchAnd* (*MatchNot* (*Match* ((*Src-Ports* (*L4Ports TCP* [(*1*,*2*)]))):: *32 common-primitive*))) (*Match* (*Src-Ports* (*L4Ports TCP* [(*1*,*2*)]))))
value *normalize-primitive-extract* (*is-Src*, *src-sel*) *Src ipt-iprange-compress* (*MatchAnd* (*MatchNot* (*Match* (*Src* (*IpAddrNetmask* (*10*::*ipv4addr*) *2*)))) (*Match* (*Src-Ports* (*L4Ports TCP* [(*1*,*2*)])))))
value *normalize-primitive-extract* (*is-Src*, *src-sel*) *Src ipt-iprange-compress* (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (*10*::*ipv4addr*) *2*))) (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask 10 8*))) (*Match* (*Src-Ports* (*L4Ports TCP* [(*1*,*2*)]))))))

value *normalize-primitive-extract* (*is-Src*, *src-sel*) *Src ipt-iprange-compress* (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (*10*::*ipv4addr*) *2*))) (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask 192 8*))) (*Match* (*Src-Ports* (*L4Ports TCP* [(*1*,*2*)]))))))

definition *normalize-src-ips* :: *'i*::*len common-primitive match-expr* ⇒ *'i common-primitive match-expr list* **where**
normalize-src-ips = *normalize-primitive-extract* (*common-primitive.is-Src*, *src-sel*) *common-primitive.Src ipt-iprange-compress*

lemma *ipt-iprange-compress-src-matching*: *match-list* (*common-matcher*, *α*) (*map* (*Match* ∘ *Src*) (*ipt-iprange-compress ml*)) *a p* ⟷ *matches* (*common-matcher*, *α*) (*alist-and* (*NegPos-map Src ml*)) *a p*
proof −
have *matches* (*common-matcher*, *α*) (*alist-and* (*NegPos-map common-primitive.Src ml*)) *a p* ⟷ (∀ *m* ∈ *set* (*getPos ml*). *matches* (*common-matcher*, *α*) (*Match* (*Src m*)) *a p*) ∧ (∀ *m* ∈ *set* (*getNeg ml*). *matches* (*common-matcher*, *α*) (*MatchNot* (*Match* (*Src m*))) *a p*)
by(*induction ml rule*: *alist-and.induct*) (*auto simp add*: *bunch-of-lemmata-about-matches*)
also have ... ⟷ *p-src p* ∈ (⋂ *ip* ∈ *set* (*getPos ml*). *ipt-iprange-to-set ip*) − (⋃ *ip* ∈ *set* (*getNeg ml*). *ipt-iprange-to-set ip*)
by(*simp add*: *match-simplematcher-SrcDst match-simplematcher-SrcDst-not*)
also have ... ⟷ *p-src p* ∈ (⋃ *ip* ∈ *set* (*ipt-iprange-compress ml*). *ipt-iprange-to-set ip*) **using** *ipt-iprange-compress*
by *blast*
also have ... ⟷ (∃ *ip* ∈ *set* (*ipt-iprange-compress ml*). *matches* (*common-matcher*, *α*) (*Match* (*Src ip*)) *a p*)

by(*simp add*: *match-simplematcher-SrcDst*)
finally show *?thesis* **using** *match-list-matches* **by** *fastforce*
qed
lemma *normalize-src-ips*: *normalized-nnf-match m* $\Longrightarrow$
match-list (*common-matcher*, α) (*normalize-src-ips m*) *a p* = *matches* (*common-matcher*, α) *m a p*
unfolding *normalize-src-ips-def*
using *normalize-primitive-extract*[*OF - wf-disc-sel-common-primitive*(*3*), **where** *f*=*ipt-iprange-compress* **and** γ=(*common-matcher*, α)]
ipt-iprange-compress-src-matching **by** *blast*

lemma *normalize-src-ips-normalized-n-primitive*: *normalized-nnf-match m* $\Longrightarrow$
$\forall$ *m'* $\in$ *set* (*normalize-src-ips m*). *normalized-src-ips m'*
unfolding *normalize-src-ips-def*
unfolding *normalized-src-ips-def2*
apply(*rule normalize-primitive-extract-normalizes-n-primitive*[*OF - wf-disc-sel-common-primitive*(*3*)])
by(*simp-all add*: *ipt-iprange-compress-normalized-IpAddrNetmask*)

definition *normalize-dst-ips* :: *'i*::*len common-primitive match-expr* $\Rightarrow$ *'i common-primitive match-expr list* **where**
normalize-dst-ips = *normalize-primitive-extract* (*common-primitive.is-Dst*, *dst-sel*)
common-primitive.Dst ipt-iprange-compress

lemma *ipt-iprange-compress-dst-matching*: *match-list* (*common-matcher*, α) (*map* (*Match* $\circ$ *Dst*) (*ipt-iprange-compress ml*)) *a p* $\longleftrightarrow$
matches (*common-matcher*, α) (*alist-and* (*NegPos-map Dst ml*)) *a p*
proof $-$
have *matches* (*common-matcher*, α) (*alist-and* (*NegPos-map common-primitive.Dst ml*)) *a p* $\longleftrightarrow$
($\forall$ *m* $\in$ *set* (*getPos ml*). *matches* (*common-matcher*, α) (*Match* (*Dst m*)) *a p*) $\wedge$
($\forall$ *m* $\in$ *set* (*getNeg ml*). *matches* (*common-matcher*, α) (*MatchNot* (*Match* (*Dst m*))) *a p*)
by(*induction ml rule*: *alist-and.induct*) (*auto simp add*: *bunch-of-lemmata-about-matches*)
also have ... $\longleftrightarrow$ *p-dst p* $\in$ ($\bigcap$ *ip* $\in$ *set* (*getPos ml*). *ipt-iprange-to-set ip*) $-$ ($\bigcup$ *ip* $\in$ *set* (*getNeg ml*). *ipt-iprange-to-set ip*)
by(*simp add*: *match-simplematcher-SrcDst match-simplematcher-SrcDst-not*)
also have ... $\longleftrightarrow$ *p-dst p* $\in$ ($\bigcup$ *ip* $\in$ *set* (*ipt-iprange-compress ml*). *ipt-iprange-to-set ip*) **using** *ipt-iprange-compress* **by** *blast*
also have ... $\longleftrightarrow$ ($\exists$ *ip* $\in$ *set* (*ipt-iprange-compress ml*). *matches* (*common-matcher*, α) (*Match* (*Dst ip*)) *a p*)
by(*simp add*: *match-simplematcher-SrcDst*)
finally show *?thesis* **using** *match-list-matches* **by** *fastforce*
qed
lemma *normalize-dst-ips*: *normalized-nnf-match m* $\Longrightarrow$
match-list (*common-matcher*, α) (*normalize-dst-ips m*) *a p* = *matches* (*common-matcher*, α) *m a p*
unfolding *normalize-dst-ips-def*

using *normalize-primitive-extract*[*OF* - *wf-disc-sel-common-primitive*(*4*), **where** *f*=*ipt-iprange-compress* **and** γ=(*common-matcher*, α)]
ipt-iprange-compress-dst-matching **by** *blast*

Normalizing the dst ips preserves the normalized src ips

lemma *normalized-nnf-match m* $\Longrightarrow$ *normalized-src-ips m* $\Longrightarrow$ $\forall$ *mn*$\in$*set* (*normalize-dst-ips m*). *normalized-src-ips mn*
unfolding *normalize-dst-ips-def normalized-src-ips-def2*
by(*rule normalize-primitive-extract-preserves-unrelated-normalized-n-primitive*)(*simp-all add*: *wf-disc-sel-common-primitive*)

lemma *normalize-dst-ips-normalized-n-primitive*: *normalized-nnf-match m* $\Longrightarrow$
$\forall$ *m'* $\in$ *set* (*normalize-dst-ips m*). *normalized-dst-ips m'*
unfolding *normalize-dst-ips-def normalized-dst-ips-def2*
by(*rule normalize-primitive-extract-normalizes-n-primitive*[*OF* - *wf-disc-sel-common-primitive*(*4*)])(*simp-all add*: *ipt-iprange-compress-normalized-IpAddrNetmask*)

end
theory *Interfaces-Normalize*
imports *Common-Primitive-Lemmas*
begin

27.7 Optimizing interfaces in match expressions

definition *compress-interfaces* :: *iface negation-type list* $\Rightarrow$ (*iface list* $\times$ *iface list*) *option* **where**
compress-interfaces ifces $\equiv$ *case* (*compress-pos-interfaces* (*getPos ifces*))
of None $\Rightarrow$ *None*
| *Some i* $\Rightarrow$ *if*
$\exists$ *negated-ifce* $\in$ *set* (*getNeg ifces*). *iface-subset i negated-ifce*
then
None
else if
$\neg$ *iface-is-wildcard i*
then
Some ([*i*], [])
else
Some ((*if i* = *ifaceAny then* [] *else* [*i*]), *getNeg ifces*)

context
begin
private lemma *compress-interfaces-None*:
assumes *generic*: *primitive-matcher-generic* β
shows
compress-interfaces ifces = *None* $\Longrightarrow$ $\neg$ *matches* (β, α) (*alist-and* (*NegPos-map IIface ifces*)) *a p*

compress-interfaces ifces = None ⟹ ¬ matches (β, α) (alist-and (NegPos-map OIface ifces)) a p
apply(*simp-all add*: *compress-interfaces-def*)
apply(*simp-all add*: *nt-match-list-matches*[*symmetric*] *nt-match-list-simp*)
apply(*simp-all add*: *NegPos-map-simps primitive-matcher-generic.Iface-single*[*OF generic*]
primitive-matcher-generic.Iface-single-not[*OF generic*])
apply(*case-tac* [!] *compress-pos-interfaces* (*getPos ifces*))
apply(*simp-all*)
apply(*drule-tac p-i=p-iiface p* **in** *compress-pos-interfaces-None*)
apply(*simp*; *fail*)
apply(*drule-tac p-i=p-iiface p* **in** *compress-pos-interfaces-Some*)
apply(*simp split*:*if-split-asm*)
using *iface-subset* **apply** *blast*
apply(*drule-tac p-i=p-oiface p* **in** *compress-pos-interfaces-None*)
apply(*simp*; *fail*)
apply(*drule-tac p-i=p-oiface p* **in** *compress-pos-interfaces-Some*)
apply(*simp split*:*if-split-asm*)
using *iface-subset* **by** *blast*

private lemma *compress-interfaces-Some*:
assumes *generic*: *primitive-matcher-generic β*
shows
compress-interfaces ifces = Some (*i-pos*, *i-neg*) ⟹
matches (*β*, *α*) (*alist-and* (*NegPos-map IIface* ((*map Pos i-pos*)@(*map Neg i-neg*)))) *a p* ⟷
matches (*β*, *α*) (*alist-and* (*NegPos-map IIface ifces*)) *a p*
compress-interfaces ifces = Some (*i-pos*, *i-neg*) ⟹
matches (*β*, *α*) (*alist-and* (*NegPos-map OIface* ((*map Pos i-pos*)@(*map Neg i-neg*)))) *a p* ⟷
matches (*β*, *α*) (*alist-and* (*NegPos-map OIface ifces*)) *a p*
apply(*simp-all add*: *compress-interfaces-def*)
apply(*simp-all add*: *bunch-of-lemmata-about-matches*(*1*) *alist-and-append NegPos-map-append*)
apply(*simp-all add*: *nt-match-list-matches*[*symmetric*] *nt-match-list-simp*)
apply(*simp-all add*: *NegPos-map-simps primitive-matcher-generic.Iface-single*[*OF generic*]
primitive-matcher-generic.Iface-single-not[*OF generic*])
apply(*case-tac* [!] *compress-pos-interfaces* (*getPos ifces*))
apply(*simp-all*)
apply(*drule-tac p-i=p-iiface p* **in** *compress-pos-interfaces-Some*)
apply(*simp split*:*if-split-asm add*: *match-ifaceAny*)
unfolding *iface-is-wildcard-def* **using** *iface-subset match-iface-case-nowildcard*
apply (*metis* (*no-types*, *lifting*) *Collect-mono-iff iface.collapse list.distinct*(*1*) *list.set-cases list.set-intros*(*1*) *set-ConsD*)
apply *force*
apply(*drule-tac p-i=p-oiface p* **in** *compress-pos-interfaces-Some*)
apply(*simp split*:*if-split-asm add*: *match-ifaceAny*)
unfolding *iface-is-wildcard-def iface-subset* **using** *match-iface-case-nowildcard*

apply (*metis iface.collapse list.distinct*(*1*) *list.inject list.set-cases list.set-intros*(*1*) *mem-Collect-eq subsetI*)
by *force*

definition *compress-normalize-input-interfaces* :: *'i*::*len common-primitive match-expr* ⇒ *'i common-primitive match-expr option* **where**
compress-normalize-input-interfaces m ≡ *compress-normalize-primitive* (*is-Iiface*, *iiface-sel*) *IIface compress-interfaces m*

lemma *compress-normalize-input-interfaces-Some*:
assumes *generic*: *primitive-matcher-generic* β
and *normalized-nnf-match m* **and** *compress-normalize-input-interfaces m* = *Some m'*
shows *matches* (β, α) *m' a p* ⟷ *matches* (β, α) *m a p*
apply(*rule compress-normalize-primitive-Some*[*OF assms*(*2*) *wf-disc-sel-common-primitive*(*5*)])
using *assms*(*3*) **apply**(*simp add*: *compress-normalize-input-interfaces-def*; *fail*)
using *compress-interfaces-Some*[*OF generic*] **by** *simp*

lemma *compress-normalize-input-interfaces-None*:
assumes *generic*: *primitive-matcher-generic* β
and *normalized-nnf-match m* **and** *compress-normalize-input-interfaces m* = *None*
shows ¬ *matches* (β, α) *m a p*
apply(*rule compress-normalize-primitive-None*[*OF assms*(*2*) *wf-disc-sel-common-primitive*(*5*)])
using *assms*(*3*) **apply**(*simp add*: *compress-normalize-input-interfaces-def*; *fail*)
using *compress-interfaces-None*[*OF generic*] **by** *simp*

lemma *compress-normalize-input-interfaces-nnf*: *normalized-nnf-match m* ⟹ *compress-normalize-input-interfaces m* = *Some m'* ⟹
normalized-nnf-match m'
unfolding *compress-normalize-input-interfaces-def*
using *compress-normalize-primitive-nnf*[*OF wf-disc-sel-common-primitive*(*5*)]
by *blast*

lemma *compress-normalize-input-interfaces-not-introduces-Iiface*:
¬ *has-disc is-Iiface m* ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-input-interfaces m* = *Some m'* ⟹
¬ *has-disc is-Iiface m'*
apply(*simp add*: *compress-normalize-input-interfaces-def*)
apply(*drule compress-normalize-primitive-not-introduces-C*[**where** *m*=*m* **and** *C'*=*IIface*])
apply(*simp-all add*: *wf-disc-sel-common-primitive*(*5*))
by(*simp add*: *compress-interfaces-def iface-is-wildcard-ifaceAny*)

lemma *compress-normalize-input-interfaces-not-introduces-Iiface-negated*:
assumes *notdisc*: ¬ *has-disc-negated is-Iiface False m*
and *nm*: *normalized-nnf-match m*
and *some*: *compress-normalize-input-interfaces m* = *Some m'*

shows ¬ *has-disc-negated is-Iface False m′*
apply(*rule compress-normalize-primitive-not-introduces-C-negated*[*OF notdisc wf-disc-sel-common-primitive*(*5*) *nm*])
using *some* **apply**(*simp add*: *compress-normalize-input-interfaces-def*)
by(*simp add*: *compress-interfaces-def split*: *option.split-asm if-split-asm*)

lemma *compress-normalize-input-interfaces-hasdisc*:
¬ *has-disc disc m* ⟹ (∀ *a*. ¬ *disc* (*IIface a*)) ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-input-interfaces m* = *Some m′* ⟹
normalized-nnf-match m′ ∧ ¬ *has-disc disc m′*
unfolding *compress-normalize-input-interfaces-def*
using *compress-normalize-primitive-hasdisc*[*OF - wf-disc-sel-common-primitive*(*5*)]
by *blast*

lemma *compress-normalize-input-interfaces-hasdisc-negated*:
¬ *has-disc-negated disc neg m* ⟹ (∀ *a*. ¬ *disc* (*IIface a*)) ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-input-interfaces m* = *Some m′* ⟹
normalized-nnf-match m′ ∧ ¬ *has-disc-negated disc neg m′*
unfolding *compress-normalize-input-interfaces-def*
using *compress-normalize-primitive-hasdisc-negated*[*OF - wf-disc-sel-common-primitive*(*5*)]
by *blast*

lemma *compress-normalize-input-interfaces-preserves-normalized-n-primitive*:
normalized-n-primitive (*disc*, *sel*) *P m* ⟹ (∀ *a*. ¬ *disc* (*IIface a*)) ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-input-interfaces m* = *Some m′* ⟹
normalized-nnf-match m′ ∧ *normalized-n-primitive* (*disc*, *sel*) *P m′*
unfolding *compress-normalize-input-interfaces-def*
using *compress-normalize-primitve-preserves-normalized-n-primitive*[*OF - wf-disc-sel-common-primitive*(*5*)]
by *blast*

value[*code*] *compress-normalize-input-interfaces*
(*MatchAnd* (*MatchAnd* (*MatchAnd* (*Match* ((*IIface* (*Iface* ″*eth+*″)::*32 common-primitive*))) (*MatchNot* (*Match* (*IIface* (*Iface* ″*eth4*″)))))) (*Match* (*IIface* (*Iface* ″*eth1*″))))
(*Match* (*Prot* (*Proto TCP*))))

value[*code*] *compress-normalize-input-interfaces* (*MatchAny*:: *32 common-primitive match-expr*)

definition *compress-normalize-output-interfaces* :: ′*i*::*len common-primitive match-expr* ⇒ ′*i common-primitive match-expr option* **where**

compress-normalize-output-interfaces m ≡ compress-normalize-primitive (is-Oiface, oiface-sel) OIface compress-interfaces m

lemma *compress-normalize-output-interfaces-Some*:
assumes *generic*: *primitive-matcher-generic β*
and *normalized-nnf-match m* **and** *compress-normalize-output-interfaces m = Some m′*
shows *matches (β, α) m′ a p ⟷ matches (β, α) m a p*
apply(*rule compress-normalize-primitive-Some*[*OF assms*(*2*) *wf-disc-sel-common-primitive*(*6*)])
using *assms*(*3*) **apply**(*simp add*: *compress-normalize-output-interfaces-def*; *fail*)
using *compress-interfaces-Some*[*OF generic*] **by** *simp*

lemma *compress-normalize-output-interfaces-None*:
assumes *generic*: *primitive-matcher-generic β*
and *normalized-nnf-match m* **and** *compress-normalize-output-interfaces m = None*
shows ¬ *matches (β, α) m a p*
apply(*rule compress-normalize-primitive-None*[*OF assms*(*2*) *wf-disc-sel-common-primitive*(*6*)])
using *assms*(*3*) **apply**(*simp add*: *compress-normalize-output-interfaces-def*; *fail*)
using *compress-interfaces-None*[*OF generic*] **by** *simp*

lemma *compress-normalize-output-interfaces-nnf*: *normalized-nnf-match m ⟹ compress-normalize-output-interfaces m = Some m′ ⟹*
normalized-nnf-match m′
unfolding *compress-normalize-output-interfaces-def*
using *compress-normalize-primitive-nnf*[*OF wf-disc-sel-common-primitive*(*6*)]
by *blast*

lemma *compress-normalize-output-interfaces-not-introduces-Oiface*:
¬ *has-disc is-Oiface m ⟹ normalized-nnf-match m ⟹ compress-normalize-output-interfaces m = Some m′ ⟹*
¬ *has-disc is-Oiface m′*
apply(*simp add*: *compress-normalize-output-interfaces-def*)
apply(*drule compress-normalize-primitive-not-introduces-C*[**where** *m*=*m* **and** *C′*=*OIface*])
apply(*simp-all add*: *wf-disc-sel-common-primitive*(*6*))
by(*simp add*: *compress-interfaces-def iface-is-wildcard-ifaceAny*)

lemma *compress-normalize-output-interfaces-not-introduces-Oiface-negated*:
assumes *notdisc*: ¬ *has-disc-negated is-Oiface False m*
and *nm*: *normalized-nnf-match m*
and *some*: *compress-normalize-output-interfaces m = Some m′*
shows ¬ *has-disc-negated is-Oiface False m′*
apply(*rule compress-normalize-primitive-not-introduces-C-negated*[*OF notdisc wf-disc-sel-common-primitive*(*6*) *nm*])
using *some* **apply**(*simp add*: *compress-normalize-output-interfaces-def*)
by(*simp add*: *compress-interfaces-def split*: *option.split-asm if-split-asm*)

lemma *compress-normalize-output-interfaces-hasdisc*:
$\neg$ *has-disc disc m* $\Longrightarrow$ ($\forall$ *a*. $\neg$ *disc* (*OIface a*)) $\Longrightarrow$ *normalized-nnf-match m*
$\Longrightarrow$ *compress-normalize-output-interfaces m* = *Some m'* $\Longrightarrow$
normalized-nnf-match m' $\wedge$ $\neg$ *has-disc disc m'*
unfolding *compress-normalize-output-interfaces-def*
using *compress-normalize-primitive-hasdisc*[*OF - wf-disc-sel-common-primitive*(*6*)]
by *blast*

lemma *compress-normalize-output-interfaces-hasdisc-negated*:
$\neg$ *has-disc-negated disc neg m* $\Longrightarrow$ ($\forall$ *a*. $\neg$ *disc* (*OIface a*)) $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *compress-normalize-output-interfaces m* = *Some m'* $\Longrightarrow$
normalized-nnf-match m' $\wedge$ $\neg$ *has-disc-negated disc neg m'*
unfolding *compress-normalize-output-interfaces-def*
using *compress-normalize-primitive-hasdisc-negated*[*OF - wf-disc-sel-common-primitive*(*6*)]
by *blast*

lemma *compress-normalize-output-interfaces-preserves-normalized-n-primitive*:
normalized-n-primitive (*disc*, *sel*) *P m* $\Longrightarrow$ ($\forall$ *a*. $\neg$ *disc* (*OIface a*)) $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *compress-normalize-output-interfaces m* = *Some m'* $\Longrightarrow$
normalized-nnf-match m' $\wedge$ *normalized-n-primitive* (*disc*, *sel*) *P m'*
unfolding *compress-normalize-output-interfaces-def*
using *compress-normalize-primitve-preserves-normalized-n-primitive*[*OF - wf-disc-sel-common-primitive*(*6*)]
by *blast*

end

end

28 Word Upto

theory *Word-Upto*
imports *Main*
IP-Addresses.Hs-Compat
Word-Lib.Word-Lemmas
begin

definition *word-range* :: ‹'*a*::*len word* $\Rightarrow$ '*a word* $\Rightarrow$ '*a word list*›
where ‹*word-range v w* = *sorted-list-of-set* {*v*..*w*}›

lemma *word-range-code* [*code*]:
‹*word-range v w* = (*if v* = *w then* [*v*] *else if v* < *w then v* # *word-range* (*v* + *1*) *w else* [])›
proof (*cases* ‹*v* < *w*›)
case *True*
then have ‹{*v*..*w*} = *insert v* {*v* + *1*..*w*}›
by (*auto simp add*: *eq-neg-iff-add-eq-0 less-is-non-zero-p1 word-Suc-le*)

moreover have ‹$v \notin \{v + 1..w\}$›
using *True* **by** (*auto simp add*: *plus-1-less*)
ultimately show *?thesis*
using *True* **by** (*auto simp add*: *word-range-def plus-1-less intro*!: *insort-is-Cons*)
next
case *False*
then show *?thesis*
by (*simp add*: *word-range-def*)
qed

lemma *word-interval-eq-set-word-range* [*code-unfold*]:
‹$\{v..w\} = set\ (word\text{-}range\ v\ w)$›
by (*simp add*: *word-range-def*)

Enumerate a range of machine words.

enumerate from the back (inefficient)

function *word-upto* :: *'a word* ⇒ *'a word* ⇒ (*'a*::*len*) *word list* **where**
word-upto a b = (*if a* = *b then* [*a*] *else word-upto a* (*b* − *1*) @ [*b*])
by *pat-completeness auto*

termination *word-upto*
apply(*relation measure* (*unat* ∘ *uncurry* (−) ∘ *prod.swap*))
apply(*rule wf-measure*; *fail*)
apply(*simp*)
apply(*subgoal-tac unat* (*b* − *a* − *1*) < *unat* (*b* − *a*))
apply(*simp add*: *diff-right-commute*; *fail*)
apply(*rule measure-unat*)
apply *auto*
done

declare *word-upto.simps*[*simp del*]

enumerate from the front (more inefficient)

function *word-upto'* :: *'a word* ⇒ *'a word* ⇒ (*'a*::*len*) *word list* **where**
word-upto' a b = (*if a* = *b then* [*a*] *else a* # *word-upto'* (*a* + *1*) *b*)
by *pat-completeness auto*

termination *word-upto'*
apply(*relation measure* (λ (*a*, *b*). *unat* (*b* − *a*)))
apply(*rule wf-measure*; *fail*)
apply(*simp*)
apply(*subgoal-tac unat* (*b* − *a* − *1*) < *unat* (*b* − *a*))
apply (*simp add*: *diff-diff-add*; *fail*)
apply(*rule measure-unat*)
apply *auto*
done

declare *word-upto'.simps*[*simp del*]

lemma *word-upto-cons-front*[*code*]:
word-upto a b = word-upto' a b
proof(*induction a b rule*:*word-upto'.induct*)
case (*1 a b*)
have *hlp1*: $a \neq b \Longrightarrow a \# word\text{-}upto\ (a + 1)\ b = word\text{-}upto\ a\ b$
apply(*induction a b rule*:*word-upto.induct*)
apply *simp*
apply(*subst*(*1*) *word-upto.simps*)
apply(*simp*)
apply *safe*
apply(*subst*(*1*) *word-upto.simps*)
apply (*simp*)
apply(*subst*(*1*) *word-upto.simps*)
apply (*simp*; *fail*)
apply(*case-tac* $a \neq b - 1$)
apply(*simp*)
apply (*metis Cons-eq-appendI word-upto.simps*)
apply(*simp*)
done

from *1*[*symmetric*] **show** *?case*
apply(*cases* $a = b$)
subgoal
apply(*subst word-upto.simps*)
apply(*subst word-upto'.simps*)
by(*simp*)
apply(*subst word-upto'.simps*)
by(*simp add*: *hlp1*)
qed

lemma *word-upto-set-eq*: $a \leq b \Longrightarrow x \in set\ (word\text{-}upto\ a\ b) \longleftrightarrow a \leq x \wedge x \leq b$
proof
show $a \leq b \Longrightarrow x \in set\ (word\text{-}upto\ a\ b) \Longrightarrow a \leq x \wedge x \leq b$
apply(*induction a b rule*: *word-upto.induct*)
apply(*case-tac* $a = b$)
apply(*subst*(*asm*) *word-upto.simps*)
apply(*simp*; *fail*)
apply(*subst*(*asm*) *word-upto.simps*)
apply(*simp*)
apply(*erule disjE*)
apply(*simp*; *fail*)
proof(*goal-cases*)

```
    case (1 a b)
    from 1(2−3) have b ≠ 0 by force
    from 1(2,3) have a ≤ b − 1
      by (simp add: word-le-minus-one-leq)
      from 1(1)[OF this 1(4)] show ?case by (metis dual-order.trans 1(2,3)
less-imp-le measure-unat word-le-0-iff word-le-nat-alt)
   qed
next
 show a ≤ x ∧ x ≤ b ⟹ x ∈ set (word-upto a b)
   apply(induction a b rule: word-upto.induct)
   apply(case-tac a = b)
    apply(subst word-upto.simps)
    apply(simp; force)
   apply(subst word-upto.simps)
   apply(simp)
   apply(case-tac x = b)
    apply(simp;fail)
   proof(goal-cases)
     case (1 a b)
     from 1(2−4) have b ≠ 0 by force
     from 1(2,4) have x ≤ b − 1
       using le-step-down-word by auto
     from 1(1) this show ?case by simp
   qed
qed

lemma word-upto-distinct-hlp: a ≤ b ⟹ a ≠ b ⟹ b ∉ set (word-upto a (b −
1))
  apply(rule ccontr, unfold not-not)
  apply(subgoal-tac a ≤ b − 1)
   apply(drule iffD1[OF word-upto-set-eq[of a b −1 b]])
    apply(simp add: word-upto.simps)
   apply(subgoal-tac b ≠ 0)
    apply(meson leD measure-unat word-le-nat-alt)
  apply(blast intro: iffD1[OF word-le-0-iff])
 using le-step-down-word apply blast
done

lemma distinct-word-upto: a ≤ b ⟹ distinct (word-upto a b)
apply(induction a b rule: word-upto.induct)
apply(case-tac a = b)
 apply(subst word-upto.simps)
 apply(simp; force)
apply(subst word-upto.simps)
apply(case-tac a ≤ b − 1)
 apply(simp)
 apply(rule word-upto-distinct-hlp; simp)
apply(simp)
  apply(rule ccontr)
```

apply (*simp add*: *not-le antisym word-minus-one-le-leq*)
done

lemma *word-upto-eq-upto*: $s \leq e \Longrightarrow e \leq unat\ (max\text{-}word :: 'l\ word) \Longrightarrow$
word-upto ((*of-nat* :: *nat* ⇒ ('*l* :: *len*) *word*) *s*) (*of-nat e*) = *map of-nat* (*upt s* (*Suc e*))
proof(*induction e*)
let *?mwon* = *of-nat* :: *nat* ⇒ '*l word*
let *?mmw* = *max-word* :: '*l word*
case (*Suc e*)
show *?case*
proof(*cases ?mwon s* = *?mwon* (*Suc e*))
case *True*
have *s* = *Suc e* **using** *le-unat-uoi Suc.prems True* **by** *metis*
with *True* **show** *?thesis* **by**(*subst word-upto.simps*) (*simp*)
next
case *False*
hence *le*: *s* ≤ *e* **using** *le-SucE Suc.prems* **by** *blast*
have *lm*: *e* ≤ *unat ?mmw* **using** *Suc.prems* **by** *simp*
have *sucm*: (*of-nat* :: *nat* ⇒ ('*l* :: *len*) *word*) (*Suc e*) − *1* = *of-nat e* **using** *Suc.prems*(*2*) **by** *simp*
note *mIH* = *Suc.IH*[*OF le lm*]
show *?thesis* **by**(*subst word-upto.simps*) (*simp add*: *False*[*simplified*] *Suc.prems mIH sucm*)
qed
qed(*simp add*: *word-upto.simps*)

lemma *word-upto-alt*: (*a* :: ('*l* :: *len*) *word*) ≤ *b* ⟹
word-upto a b = *map of-nat* (*upt* (*unat a*) (*Suc* (*unat b*)))
proof −
let *?mmw* = − *1* :: '*l word*
assume *le*: *a* ≤ *b*
hence *nle*: *unat a* ≤ *unat b* **by**(*unat-arith*)
have *lem*: *unat b* ≤ *unat ?mmw* **by** (*simp add*: *word-unat-less-le*)
then show *word-upto a b* = *map of-nat* [*unat a*..<*Suc* (*unat b*)]
using *word-upto-eq-upto* [*OF nle lem*] **by** *simp*
qed

lemma *word-upto-upt*:
word-upto a b = (*if a* ≤ *b then map of-nat* (*upt* (*unat a*) (*Suc* (*unat b*))) *else word-upto a b*)
using *word-upto-alt* **by** *metis*

lemma *sorted-word-upto*:
fixes *a b* :: ('*l* :: *len*) *word*
assumes *a* ≤ *b*
shows *sorted* (*word-upto a b*)
proof −

define *m* **and** *n* **where** ‹*m* = *unat a*› **and** ‹*n* = *Suc* (*unat b*)›
moreover have ‹*sorted-wrt* (λ*x y*. (*word-of-nat x* :: *'l word*) ≤ *word-of-nat y*) [*m*..<*n*]›
proof (*rule sorted-wrt-mono-rel* [*of* - ‹(≤)›])
show ‹*sorted* [*m*..<*n*]›
by *simp*
fix *r s*
assume ‹*r* ∈ *set* [*m*..<*n*]› ‹*s* ∈ *set* [*m*..<*n*]› ‹*r* ≤ *s*›
then have ‹*m* ≤ *r*› ‹*s* < *n*›
by *simp-all*
then have ‹*take-bit LENGTH*('*l*) *s* = *s*›
by (*auto simp add*: *m-def n-def less-Suc-eq-le unsigned-of-nat dest*: *le-unat-uoi*)
with ‹*r* ≤ *s*› **show** ‹(*word-of-nat r* :: *'l word*) ≤ *word-of-nat s*›
apply (*simp add*: *of-nat-word-less-eq-iff*)
using *take-bit-nat-less-eq-self* **apply** (*rule order-trans*)
apply *assumption*
done
qed
then have ‹*sorted* (*map word-of-nat* [*m*..<*n*] :: *'l word list*)›
by (*simp add*: *sorted-map*)
ultimately have ‹*sorted* (*map of-nat* [*unat a*..<*Suc* (*unat b*)] :: *'l word list*)›
by *simp*
with *assms* **show** *?thesis*
by (*simp only*: *word-upto-alt*)
qed

end
theory *Protocols-Normalize*
imports *Common-Primitive-Lemmas*
../Common/Word-Upto
begin

29 Optimizing Protocols

30 Optimizing protocols in match expressions

fun *compress-pos-protocols* :: *protocol list* ⇒ *protocol option* **where**
compress-pos-protocols [] = *Some ProtoAny* |
compress-pos-protocols [*p*] = *Some p* |
compress-pos-protocols (*p1*#*p2*#*ps*) = (*case simple-proto-conjunct p1 p2 of None* ⇒ *None* | *Some p* ⇒ *compress-pos-protocols* (*p*#*ps*))

lemma *compress-pos-protocols-Some*: *compress-pos-protocols ps* = *Some proto* ⟹
match-proto proto p-prot ⟷ (∀ *p* ∈ *set ps*. *match-proto p p-prot*)
proof(*induction ps rule*: *compress-pos-protocols.induct*)
case (*3 p1 p2 pps*) **thus** *?case*
apply(*cases simple-proto-conjunct p1 p2*)

apply(*simp*; *fail*)
using *simple-proto-conjunct-Some* **by**(*simp*)
qed(*simp*)+

lemma *compress-pos-protocols-None*: *compress-pos-protocols ps* = *None* ⟹
¬ (∀ *proto* ∈ *set ps*. *match-proto proto p-prot*)
proof(*induction ps rule*: *compress-pos-protocols.induct*)
case (*3 i1 i2 iis*) **thus** *?case*
apply(*cases simple-proto-conjunct i1 i2*)
apply(*simp-all*)
using *simple-proto-conjunct-None* **apply** *blast*
using *simple-proto-conjunct-Some* **by** *blast*
qed(*simp*)+

lemma *simple-proto-conjunct* (*Proto p1*) (*Proto p2*) ≠ *None* ⟹ ∀ *pkt*. *match-proto* (*Proto p1*) *pkt* ⟷ *match-proto* (*Proto p2*) *pkt*
apply(*subgoal-tac p1* = *p2*)
apply(*simp*)
apply(*simp split*: *if-split-asm*)
done
lemma *simple-proto-conjunct p1* (*Proto p2*) ≠ *None* ⟹ ∀ *pkt*. *match-proto* (*Proto p2*) *pkt* ⟶ *match-proto p1 pkt*
apply(*cases p1*)
apply(*simp*)
apply(*simp split*: *if-split-asm*)
done

definition *compress-protocols* :: *protocol negation-type list* ⇒ (*protocol list* × *protocol list*) *option* **where**
compress-protocols ps ≡ *case* (*compress-pos-protocols* (*getPos ps*))
of None ⇒ *None*
| *Some proto* ⇒ *if ProtoAny* ∈ *set* (*getNeg ps*) ∨ (∀ *p* ∈ {*0*..− *1*}. *Proto p* ∈ *set* (*getNeg ps*)) *then*
None
else if proto = *ProtoAny then*
Some ([], *getNeg ps*)
else if (∃ *p* ∈ *set* (*getNeg ps*). *simple-proto-conjunct proto p* ≠ *None*) *then*
None
else
— *proto* is a *primitive-protocol* here. This is strict equality match, e.g.
— protocol must be TCP. Thus, we can remove all negative matches!
Some ([*proto*], [])

lemma *all-proto-hlp2*: *ProtoAny* ∈ *a* ∨ (∀ *p* ∈ {*0*..− *1*}. *Proto p* ∈ *a*) ⟷

```
                    ProtoAny ∈ a ∨ a = {p. p ≠ ProtoAny}
  proof −
    have all-proto-hlp: ProtoAny ∉ a ⟹ (∀ p ∈ {0..− 1}. Proto p ∈ a) ⟷ a =
{p. p ≠ ProtoAny}
      by(auto intro: protocol.exhaust)
    thus ?thesis by blast
  qed

  lemma set-word8-word-upto: {0..(− 1 :: 8 word)} = set (word-upto 0 255)
  proof −
    have ‹0xFF = (− 1 :: 8 word)›
      by simp
    then show ?thesis
      by (simp only:) (auto simp add: word-upto-set-eq)
  qed
  lemma (∀ p ∈ {0..− 1}. Proto p ∈ set (getNeg ps)) ⟷
        ((∀ p ∈ set (word-upto 0 255). Proto p ∈ set (getNeg ps)))
    by(simp add: set-word8-word-upto)


  lemma compress-protocols-code[code]:
    compress-protocols ps = (case (compress-pos-protocols (getPos ps))
       of None ⇒ None
       | Some proto ⇒ if ProtoAny ∈ set (getNeg ps) ∨ (∀ p ∈ set (word-upto 0
255). Proto p ∈ set (getNeg ps)) then
                         None
                       else if proto = ProtoAny then
                         Some ([], getNeg ps)
                       else if (∃ p ∈ set (getNeg ps). simple-proto-conjunct proto p ≠
None) then
                         None
                       else
                         Some ([proto], [])
      )
    unfolding compress-protocols-def
    using set-word8-word-upto by presburger


  lemma compress-protocols ps = Some (ps-pos, ps-neg) ⟹
    ∃ p. ((∀ m∈set ps-pos. match-proto m p) ∧ (∀ m∈set ps-neg. ¬ match-proto m
p))
     apply(simp add: compress-protocols-def all-proto-hlp2 split: option.split-asm
if-split-asm)
     apply(subgoal-tac ∃ p. (Proto p) ∉ set ps-neg)
      apply(elim exE)
      apply(rename-tac x2 p)
      apply(rule-tac x=p in exI)
      apply(blast elim: match-proto.elims)
     apply(auto intro: protocol.exhaust)
```

done

definition *compress-normalize-protocols-step* :: *'i*::*len common-primitive match-expr* ⇒ *'i common-primitive match-expr option* **where**
compress-normalize-protocols-step m ≡ *compress-normalize-primitive* (*is-Prot*, *prot-sel*) *Prot compress-protocols m*

lemma (**in** *primitive-matcher-generic*) *compress-normalize-protocols-step-Some*:
assumes *normalized-nnf-match m* **and** *compress-normalize-protocols-step m* = *Some m'*
shows *matches* (*β*, *α*) *m' a p* ⟷ *matches* (*β*, *α*) *m a p*
proof(*rule compress-normalize-primitive-Some*[*OF assms*(*1*) *wf-disc-sel-common-primitive*(*7*), *of compress-protocols*])
show *compress-normalize-primitive* (*is-Prot*, *prot-sel*) *Prot compress-protocols m* = *Some m'*
using *assms*(*2*) **by**(*simp add*: *compress-normalize-protocols-step-def*)
next
fix *ps ps-pos ps-neg*
show *compress-protocols ps* = *Some* (*ps-pos*, *ps-neg*) ⟹
matches (*β*, *α*) (*alist-and* (*NegPos-map Prot* ((*map Pos ps-pos*)@(*map Neg ps-neg*)))) *a p* ⟷
matches (*β*, *α*) (*alist-and* (*NegPos-map Prot ps*)) *a p*
apply(*simp add*: *compress-protocols-def*)
apply(*simp add*: *bunch-of-lemmata-about-matches alist-and-append NegPos-map-append*)
apply(*simp add*: *nt-match-list-matches*[*symmetric*] *nt-match-list-simp*)
apply(*simp add*: *NegPos-map-simps Prot-single Prot-single-not*)
apply(*case-tac compress-pos-protocols* (*getPos ps*))
apply(*simp-all*)
apply(*drule-tac p-prot*=*p-proto p* **in** *compress-pos-protocols-Some*)
apply(*simp split*:*if-split-asm*)
using *simple-proto-conjunct-None* **by** *auto*
qed

lemma (**in** *primitive-matcher-generic*) *compress-normalize-protocols-step-None*:
assumes *normalized-nnf-match m* **and** *compress-normalize-protocols-step m* = *None*
shows ¬ *matches* (*β*, *α*) *m a p*
proof(*rule compress-normalize-primitive-None*[*OF assms*(*1*) *wf-disc-sel-common-primitive*(*7*), *of compress-protocols*])
show *compress-normalize-primitive* (*is-Prot*, *prot-sel*) *Prot compress-protocols m* = *None*
using *assms*(*2*) **by**(*simp add*: *compress-normalize-protocols-step-def*)
next
fix *ps*
have *if-option-Some*:
((*if P then None else Some x*) = *Some y*) = (¬*P* ∧ *x* = *y*)
for *P* **and** *x*::*protocol* **and** *y* **by** *simp*
show *compress-protocols ps* = *None* ⟹ ¬ *matches* (*β*, *α*) (*alist-and* (*NegPos-map Prot ps*)) *a p*

apply(*simp add*: *compress-protocols-def*)
apply(*simp add*: *nt-match-list-matches*[*symmetric*] *nt-match-list-simp*)
apply(*simp add*: *NegPos-map-simps Prot-single Prot-single-not*)
apply(*cases compress-pos-protocols* (*getPos ps*))
apply(*simp-all*)
apply(*drule-tac p-prot=p-proto p* **in** *compress-pos-protocols-None*)
apply(*simp*; *fail*)
apply(*drule-tac p-prot=p-proto p* **in** *compress-pos-protocols-Some*)
apply(*simp split*:*if-split-asm*)
apply *fastforce*
apply(*elim bexE exE*)
apply(*simp*)
apply(*elim simple-proto-conjunct.elims*)
apply(*simp*; *fail*)
apply(*simp*; *fail*)
using *if-option-Some* **by** *metis*
qed

lemma *compress-normalize-protocols-step-nnf*:
normalized-nnf-match m ⟹ *compress-normalize-protocols-step m* = *Some m′*
⟹
normalized-nnf-match m′
unfolding *compress-normalize-protocols-step-def*
using *compress-normalize-primitive-nnf*[*OF wf-disc-sel-common-primitive*(*7*)]
by *blast*

lemma *compress-normalize-protocols-step-not-introduces-Prot*:
¬ *has-disc is-Prot m* ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-protocols-step*
m = *Some m′* ⟹
¬ *has-disc is-Prot m′*
apply(*simp add*: *compress-normalize-protocols-step-def*)
apply(*drule compress-normalize-primitive-not-introduces-C*[**where** *m*=*m* **and**
C′=*Prot*])
apply(*simp-all add*: *wf-disc-sel-common-primitive*(*7*))
apply(*simp add*: *compress-protocols-def split*: *if-splits*)
done

lemma *compress-normalize-protocols-step-not-introduces-Prot-negated*:
assumes *notdisc*: ¬ *has-disc-negated is-Prot False m*
and *nm*: *normalized-nnf-match m*
and *some*: *compress-normalize-protocols-step m* = *Some m′*
shows ¬ *has-disc-negated is-Prot False m′*
apply(*rule compress-normalize-primitive-not-introduces-C-negated*[*OF notdisc*
wf-disc-sel-common-primitive(*7*) *nm*])
using *some* **apply**(*simp add*: *compress-normalize-protocols-step-def*)
by(*simp add*: *compress-protocols-def split*: *option.split-asm if-split-asm*)

lemma *compress-normalize-protocols-step-hasdisc*:
¬ *has-disc disc m* ⟹ (∀ *a*. ¬ *disc* (*Prot a*)) ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-protocols-step m* = *Some m'* ⟹
normalized-nnf-match m' ∧ ¬ *has-disc disc m'*
unfolding *compress-normalize-protocols-step-def*
using *compress-normalize-primitive-hasdisc*[*OF - wf-disc-sel-common-primitive*(7)]
by *blast*

lemma *compress-normalize-protocols-step-hasdisc-negated*:
¬ *has-disc-negated disc neg m* ⟹ (∀ *a*. ¬ *disc* (*Prot a*)) ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-protocols-step m* = *Some m'* ⟹
normalized-nnf-match m' ∧ ¬ *has-disc-negated disc neg m'*
unfolding *compress-normalize-protocols-step-def*
using *compress-normalize-primitive-hasdisc-negated*[*OF - wf-disc-sel-common-primitive*(7)]
by *blast*

lemma *compress-normalize-protocols-step-preserves-normalized-n-primitive*:
normalized-n-primitive (*disc*, *sel*) *P m* ⟹ (∀ *a*. ¬ *disc* (*Prot a*)) ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-protocols-step m* = *Some m'* ⟹
normalized-nnf-match m' ∧ *normalized-n-primitive* (*disc*, *sel*) *P m'*
unfolding *compress-normalize-protocols-step-def*
using *compress-normalize-primitve-preserves-normalized-n-primitive*[*OF - wf-disc-sel-common-primitive*(7)]
by *blast*

lemma *case compress-normalize-protocols-step*
(*MatchAnd* (*MatchAnd* (*MatchAnd* (*Match* ((*Prot* (*Proto TCP*)):: *32 common-primitive*)) (*MatchNot* (*Match* (*Prot* (*Proto UDP*))))) (*Match* (*IIface* (*Iface* ''*eth1*''))))
(*Match* (*Prot* (*Proto TCP*)))) *of Some ps* ⇒ *opt-MatchAny-match-expr ps*
= *MatchAnd* (*Match* (*Prot* (*Proto 6*))) (*Match* (*IIface* (*Iface* ''*eth1*''))) **by** *eval*

value[*code*] *compress-normalize-protocols-step* (*MatchAny*:: *32 common-primitive match-expr*)

30.1 Importing the matches on *primitive-protocol* from *L4Ports*

definition *import-protocols-from-ports*
:: ′*i*::*len common-primitive match-expr* ⇒ ′*i common-primitive match-expr*
where
import-protocols-from-ports m ≡
(*case primitive-extractor* (*is-Src-Ports*, *src-ports-sel*) *m of* (*srcpts*, *rst1*) ⇒
case primitive-extractor (*is-Dst-Ports*, *dst-ports-sel*) *rst1 of* (*dstpts*, *rst2*) ⇒
MatchAnd
(*MatchAnd*
(*MatchAnd*
(*andfold-MatchExp* (*map* (*Match* ∘ (*Prot* ∘ (*case-ipt-l4-ports* (λ*proto x*. *Proto*

```
proto)))) (getPos srcpts)))
      (andfold-MatchExp (map (Match ∘ (Prot ∘ (case-ipt-l4-ports (λproto x. Proto
proto)))) (getPos dstpts)))
      )
       (alist-and' (NegPos-map Src-Ports srcpts @ NegPos-map Dst-Ports dstpts))
      )
        rst2
    )
```

The *Proto* and *L4Ports* match make the following match impossible:

```
 lemma compress-normalize-protocols-step (import-protocols-from-ports
   (MatchAnd (MatchAnd (Match (Prot (Proto TCP):: 32 common-primitive))
   (Match (Src-Ports (L4Ports UDP [(22,22)])))) (Match (IIface (Iface ''eth1''))))
= None
 by eval
```

```
 lemma import-protocols-from-ports-erule: normalized-nnf-match m ⟹ P m ⟹
   (⋀srcpts rst1 dstpts rst2.
      normalized-nnf-match m ⟹
      — P m ⟹ erule consumes only first argument
      primitive-extractor (is-Src-Ports, src-ports-sel) m = (srcpts, rst1) ⟹
      primitive-extractor (is-Dst-Ports, dst-ports-sel) rst1 = (dstpts, rst2) ⟹
      normalized-nnf-match rst1 ⟹
      normalized-nnf-match rst2 ⟹
      P (MatchAnd
         (MatchAnd
           (MatchAnd
             (andfold-MatchExp
               (map (Match ∘ (Prot ∘ (case-ipt-l4-ports (λproto x. Proto proto))))
(getPos srcpts)))
             (andfold-MatchExp
               (map (Match ∘ (Prot ∘ (case-ipt-l4-ports (λproto x. Proto proto))))
(getPos dstpts))))
              (alist-and' (NegPos-map Src-Ports srcpts @ NegPos-map Dst-Ports
dstpts)))
          rst2)) ⟹
    P (import-protocols-from-ports m)
    apply(simp add: import-protocols-from-ports-def)
    apply(case-tac primitive-extractor (is-Src-Ports, src-ports-sel) m, rename-tac
srcpts rst1)
    apply(simp)
   apply(case-tac primitive-extractor (is-Dst-Ports, dst-ports-sel) rst1, rename-tac
dstpts rst2)
    apply(simp)
   apply(frule(1) primitive-extractor-correct(2)[OF - wf-disc-sel-common-primitive(1)])
   apply(frule(1) primitive-extractor-correct(2)[OF - wf-disc-sel-common-primitive(2)])
    apply simp
```

```
  done

 lemma (in primitive-matcher-generic) import-protocols-from-ports:
 assumes normalized: normalized-nnf-match m
 shows matches (β, α) (import-protocols-from-ports m) a p ⟷ matches (β, α)
m a p
 proof−
   have add-protocol:
   matches (β, α)
    (andfold-MatchExp (map (Match ∘ (Prot ∘ (case-ipt-l4-ports (λproto x. Proto
proto)))) (getPos as))) a p ∧
    matches (β, α) (alist-and (NegPos-map C as)) a p
    ⟷
    matches (β, α) (alist-and (NegPos-map C as)) a p
   if C: C = Src-Ports ∨ C = Dst-Ports for C as
    proof(induction as)
    case Nil thus ?case by(simp)
    next
    case (Cons x xs)
     show ?case
     proof(cases x)
     case Neg with Cons.IH show ?thesis
      apply(simp add: bunch-of-lemmata-about-matches)
      by blast
     next
     case (Pos portmatch)
      with Cons.IH show ?thesis
       apply(cases portmatch)
      apply(simp add: andfold-MatchExp-matches bunch-of-lemmata-about-matches)
       using Ports-single-rewrite-Prot C by blast
     qed
    qed
 from normalized show ?thesis
  apply −
  apply(erule import-protocols-from-ports-erule)
   apply(simp; fail)
 apply(subst primitive-extractor-correct(1)[OF normalized wf-disc-sel-common-primitive(1),
       where γ=(β,α) and a=a and p=p, symmetric])
   apply(simp; fail)
 apply(drule(1) primitive-extractor-correct(1)[OF - wf-disc-sel-common-primitive(2),
       where γ=(β,α) and a=a and p=p])
  apply(simp add: bunch-of-lemmata-about-matches matches-alist-and-alist-and'
alist-and-append)
   using add-protocol by blast
 qed


 lemma import-protocols-from-ports-nnf:
  normalized-nnf-match m ⟹ normalized-nnf-match (import-protocols-from-ports
```

m)
proof −
have *hlp*: ∀ *m*∈*set* (*map* (*Match* ∘ (*Prot* ∘ (*case-ipt-l4-ports* (λ*proto x*. *Proto proto*)))) *ls*).
normalized-nnf-match m **for** *ls*
apply(*induction ls*)
apply(*simp*)
apply(*rename-tac l ls*, *case-tac l*)
by(*simp*)
show *normalized-nnf-match m* ⟹ *normalized-nnf-match* (*import-protocols-from-ports m*)
apply(*rule import-protocols-from-ports-erule*)
apply(*simp-all*)
apply(*simp add*: *normalized-nnf-match-alist-and'*)
apply(*safe*)
apply(*rule andfold-MatchExp-normalized-nnf*, *simp add*: *hlp*)+
done
qed

lemma *import-protocols-from-ports-not-introduces-Prot-negated*:
normalized-nnf-match m ⟹ ¬ *has-disc-negated is-Prot False m* ⟹
¬ *has-disc-negated is-Prot False* (*import-protocols-from-ports m*)
apply(*erule*(*1*) *import-protocols-from-ports-erule*)
apply(*simp*)
apply(*intro conjI*)
using *andfold-MatchExp-not-disc-negated-mapMatch*[
where *C*=*Prot* ∘ *case-ipt-l4-ports* (λ*proto x*. *Proto proto*), *simplified*]
apply *blast*
using *andfold-MatchExp-not-disc-negated-mapMatch*[
where *C*=*Prot* ∘ *case-ipt-l4-ports* (λ*proto x*. *Proto proto*), *simplified*] **apply** *blast*
apply(*simp add*: *has-disc-negated-alist-and'*)
using *not-has-disc-negated-NegPos-map*[**where** *disc*=*is-Prot* **and** *C*=*Src-Ports*, *simplified*]
not-has-disc-negated-NegPos-map[**where** *disc*=*is-Prot* **and** *C*=*Dst-Ports*, *simplified*] **apply** *blast*
apply(*drule*(*1*) *primitive-extractor-correct*(*6*)[*OF - wf-disc-sel-common-primitive*(*1*), **where** *neg*=*False*])
apply(*drule*(*1*) *primitive-extractor-correct*(*6*)[*OF - wf-disc-sel-common-primitive*(*2*), **where** *neg*=*False*])
by *blast*

lemma *import-protocols-from-ports-hasdisc*:
normalized-nnf-match m ⟹ ¬ *has-disc disc m* ⟹ (∀ *a*. ¬ *disc* (*Prot a*)) ⟹
normalized-nnf-match (*import-protocols-from-ports m*) ∧ ¬ *has-disc disc* (*import-protocols-from-ports m*)
apply(*intro conjI*)
using *import-protocols-from-ports-nnf* **apply** *blast*

apply(*erule*(*1*) *import-protocols-from-ports-erule*)
apply(*simp*)
apply(*intro conjI*)
using *andfold-MatchExp-not-disc-mapMatch*[
where *C*=*Prot* ∘ *case-ipt-l4-ports* (λ*proto x*. *Proto proto*), *simplified*]
apply *blast*
using *andfold-MatchExp-not-disc-mapMatch*[
where *C*=*Prot* ∘ *case-ipt-l4-ports* (λ*proto x*. *Proto proto*), *simplified*] **apply** *blast*
subgoal for *srcpts rst1 dstpts rst2*
apply(*frule*(*2*) *primitive-extractor-reassemble-not-has-disc*[*OF wf-disc-sel-common-primitive*(*1*)])
apply(*subgoal-tac* ¬ *has-disc disc rst1*)
prefer *2*
apply(*drule*(*1*) *primitive-extractor-correct*(*4*)[*OF - wf-disc-sel-common-primitive*(*1*)])
apply *blast*
apply(*drule*(*2*) *primitive-extractor-reassemble-not-has-disc*[*OF wf-disc-sel-common-primitive*(*2*)])
using *has-disc-alist-and'-append* **by** *blast*
apply(*drule*(*1*) *primitive-extractor-correct*(*4*)[*OF - wf-disc-sel-common-primitive*(*1*)])
apply(*drule*(*1*) *primitive-extractor-correct*(*4*)[*OF - wf-disc-sel-common-primitive*(*2*)])
apply *blast*
done

lemma *import-protocols-from-ports-hasdisc-negated*:
¬ *has-disc-negated disc False m* ⟹ (∀ *a*. ¬ *disc* (*Prot a*)) ⟹ *normalized-nnf-match m* ⟹
normalized-nnf-match (*import-protocols-from-ports m*) ∧
¬ *has-disc-negated disc False* (*import-protocols-from-ports m*)
apply(*intro conjI*)
using *import-protocols-from-ports-nnf* **apply** *blast*
apply(*erule*(*1*) *import-protocols-from-ports-erule*)
apply(*simp*)
apply(*intro conjI*)
using *andfold-MatchExp-not-disc-negated-mapMatch*[
where *C*=*Prot* ∘ *case-ipt-l4-ports* (λ*proto x*. *Proto proto*), *simplified*]
apply *blast*
using *andfold-MatchExp-not-disc-negated-mapMatch*[
where *C*=*Prot* ∘ *case-ipt-l4-ports* (λ*proto x*. *Proto proto*), *simplified*] **apply** *blast*
subgoal for *srcpts rst1 dstpts rst2*
apply(*frule*(*2*) *primitive-extractor-reassemble-not-has-disc-negated*[*OF wf-disc-sel-common-primitive*(*1*)])
apply(*subgoal-tac* ¬ *has-disc-negated disc False rst1*)
prefer *2*
apply(*drule*(*1*) *primitive-extractor-correct*(*6*)[*OF - wf-disc-sel-common-primitive*(*1*)])
apply *blast*
apply(*drule*(*2*) *primitive-extractor-reassemble-not-has-disc-negated*[*OF wf-disc-sel-common-primitive*(*2*)])
using *has-disc-negated-alist-and'-append* **by** *blast*
apply(*drule*(*1*) *primitive-extractor-correct*(*6*)[*OF - wf-disc-sel-common-primitive*(*1*)])

apply(*drule*(*1*) *primitive-extractor-correct*(*6*)[*OF* - *wf-disc-sel-common-primitive*(*2*)])
apply *blast*
done

lemma *import-protocols-from-ports-preserves-normalized-n-primitive*:
normalized-n-primitive (*disc*, *sel*) *f m* ⟹ (∀ *a*. ¬ *disc* (*Prot a*)) ⟹ *normalized-nnf-match m* ⟹
normalized-nnf-match (*import-protocols-from-ports m*) ∧ *normalized-n-primitive* (*disc*, *sel*) *f* (*import-protocols-from-ports m*)
apply(*intro conjI*)
using *import-protocols-from-ports-nnf* **apply** *blast*
apply(*erule*(*1*) *import-protocols-from-ports-erule*)
apply(*simp*)
apply(*intro conjI*)
subgoal for *srcpts rst1 dstpts rst2*
apply(*rule andfold-MatchExp-normalized-n-primitive*)
using *normalized-n-primitive-impossible-map* **by** *blast*
subgoal for *srcpts rst1 dstpts rst2*
apply(*rule andfold-MatchExp-normalized-n-primitive*)
using *normalized-n-primitive-impossible-map* **by** *blast*
subgoal for *srcpts rst1 dstpts rst2*
apply(*frule*(*2*) *primitive-extractor-reassemble-normalized-n-primitive*[*OF wf-disc-sel-common-primitive*(*1*)
apply(*subgoal-tac normalized-n-primitive* (*disc*, *sel*) *f rst1*)
prefer *2*
apply(*drule*(*1*) *primitive-extractor-correct*(*5*)[*OF* - *wf-disc-sel-common-primitive*(*1*)])
apply *blast*
apply(*drule*(*2*) *primitive-extractor-reassemble-normalized-n-primitive*[*OF wf-disc-sel-common-primitive*(*2*)
using *normalized-n-primitive-alist-and'-append* **by** *blast*
apply(*drule*(*1*) *primitive-extractor-correct*(*5*)[*OF* - *wf-disc-sel-common-primitive*(*1*)])
apply(*drule*(*1*) *primitive-extractor-correct*(*5*)[*OF* - *wf-disc-sel-common-primitive*(*2*)])
apply *blast*
done

30.2 Putting things together

definition *compress-normalize-protocols*
:: *'i*::*len common-primitive match-expr* ⇒ *'i common-primitive match-expr option* **where**
compress-normalize-protocols m ≡ *compress-normalize-protocols-step* (*import-protocols-from-ports m*)

lemma (**in** *primitive-matcher-generic*) *compress-normalize-protocols-Some*:
assumes *normalized-nnf-match m* **and** *compress-normalize-protocols m* = *Some m'*
shows *matches* (*β*, *α*) *m' a p* ⟷ *matches* (*β*, *α*) *m a p*
using *assms* **apply**(*simp add*: *compress-normalize-protocols-def*)
by (*metis import-protocols-from-ports import-protocols-from-ports-nnf compress-normalize-protocols-step-Some*)

lemma (**in** *primitive-matcher-generic*) *compress-normalize-protocols-None*:
assumes *normalized-nnf-match m* **and** *compress-normalize-protocols m* = *None*
shows ¬ *matches* (*β*, *α*) *m a p*
using *assms* **apply**(*simp add*: *compress-normalize-protocols-def*)
by (*metis import-protocols-from-ports import-protocols-from-ports-nnf*
compress-normalize-protocols-step-None)

lemma *compress-normalize-protocols-nnf*:
normalized-nnf-match m ⟹ *compress-normalize-protocols m* = *Some m′* ⟹
normalized-nnf-match m′
apply(*simp add*: *compress-normalize-protocols-def*)
by (*metis import-protocols-from-ports-nnf compress-normalize-protocols-step-nnf*)

lemma *compress-normalize-protocols-not-introduces-Prot-negated*:
assumes *notdisc*: ¬ *has-disc-negated is-Prot False m*
and *nm*: *normalized-nnf-match m*
and *some*: *compress-normalize-protocols m* = *Some m′*
shows ¬ *has-disc-negated is-Prot False m′*
using *assms* **apply**(*simp add*: *compress-normalize-protocols-def*)
using *import-protocols-from-ports-nnf*
import-protocols-from-ports-not-introduces-Prot-negated
compress-normalize-protocols-step-not-introduces-Prot-negated **by** *auto*

lemma *compress-normalize-protocols-hasdisc*:
¬ *has-disc disc m* ⟹ (∀ *a*. ¬ *disc* (*Prot a*)) ⟹ *normalized-nnf-match m* ⟹
compress-normalize-protocols m = *Some m′* ⟹
normalized-nnf-match m′ ∧ ¬ *has-disc disc m′*
apply(*simp add*: *compress-normalize-protocols-def*)
using *import-protocols-from-ports-hasdisc*
compress-normalize-protocols-step-hasdisc **by** *blast*

lemma *compress-normalize-protocols-hasdisc-negated*:
¬ *has-disc-negated disc False m* ⟹ (∀ *a*. ¬ *disc* (*Prot a*)) ⟹
normalized-nnf-match m ⟹ *compress-normalize-protocols m* = *Some m′* ⟹
normalized-nnf-match m′ ∧ ¬ *has-disc-negated disc False m′*
apply(*simp add*: *compress-normalize-protocols-def*)
apply(*frule*(*2*) *import-protocols-from-ports-hasdisc-negated*)
using *compress-normalize-protocols-step-hasdisc-negated* **by** *blast*

lemma *compress-normalize-protocols-preserves-normalized-n-primitive*:
normalized-n-primitive (*disc*, *sel*) *P m* ⟹ (∀ *a*. ¬ *disc* (*Prot a*)) ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-protocols m* = *Some m′* ⟹
normalized-nnf-match m′ ∧ *normalized-n-primitive* (*disc*, *sel*) *P m′*
apply(*simp add*: *compress-normalize-protocols-def*)
using *import-protocols-from-ports-preserves-normalized-n-primitive*

compress-normalize-protocols-step-preserves-normalized-n-primitive **by** *blast*

lemma *case compress-normalize-protocols*
(*MatchAnd* (*MatchAnd* (*MatchAnd* (*Match* ((*Prot* (*Proto TCP*))*::* *32 common-primitive*)) (*MatchNot* (*Match* (*Prot* (*Proto UDP*))))) (*Match* (*IIface* (*Iface* *''eth1''*))))
(*Match* (*Prot* (*Proto TCP*)))) *of Some ps* ⇒ *opt-MatchAny-match-expr ps*
=
MatchAnd (*Match* (*Prot* (*Proto 6*))) (*Match* (*IIface* (*Iface* *''eth1''*))) **by** *eval*

value[*code*] *compress-normalize-protocols* (*MatchAny*:: *32 common-primitive match-expr*)

end

31 Reverse Remdups

theory *Remdups-Rev*
imports *Main*
begin

definition *remdups-rev* :: *'a list* ⇒ *'a list* **where**
remdups-rev rs ≡ *rev* (*remdups* (*rev rs*))

lemma *remdups-append*: *remdups* (*rs* @ *rs2*) = *remdups* [*r*←*rs* . *r* ∉ *set rs2*] @ *remdups rs2*
by(*induction rs arbitrary*: *rs2*) (*simp-all*)

lemma *remdups-rev-append*: *remdups-rev* (*rs* @ *rs2*) = *remdups-rev rs* @ *remdups-rev* [*r*←*rs2* . *r* ∉ *set rs*]
proof(*induction rs arbitrary*: *rs2*)
case *Cons* **thus** *?case* **by**(*simp add*: *remdups-append rev-filter remdups-rev-def*)
qed(*simp add*: *remdups-rev-def*)

lemma *remdups-rev-fst*:
remdups-rev (*r*#*rs*) = (*if r* ∈ *set rs then r*#*remdups-rev* (*removeAll r rs*) *else r*#*remdups-rev rs*)
proof −
have *1*: *r* ∉ *set rs* ⟹ *remdups-rev* (*r* # *rs*) = *r* # *remdups-rev rs*
unfolding *remdups-rev-def*
proof(*induction rs*)
case (*Cons r rs*)
{ fix *rs* **and** *rs2*::*'a list*
have *set rs* ∩ *set rs2* = {} ⟹ *remdups* (*rs* @ *rs2*) = *remdups rs* @ *remdups rs2*
by(*induction rs arbitrary*: *rs2*) (*simp-all*)
} note *h*=*this*

```
    { fix r and rs::'a list
      from h[of rev rs [r]] have r ∉ set rs ⟹ remdups (rev rs @ [r]) = remdups
(rev rs) @ [r] by simp
    }
    with Cons show ?case by fastforce
  qed(simp)

 have 2: r ∈ set rs ⟹ remdups-rev (r # rs) = r # remdups-rev (rev (removeAll
r (rev rs)))
   unfolding remdups-rev-def
   proof(induction rs)
   case Cons thus ?case
     apply(simp add: removeAll-filter-not-eq remdups-append)
     apply(safe)
      apply(simp-all)
     apply metis
     apply metis
     done
   qed(simp)

 have rev (removeAll r (rev rs)) = removeAll r rs by (simp add: removeAll-filter-not-eq
rev-filter)
 with 1 2 show ?thesis by simp
qed

lemma remdups-rev-set: set (remdups-rev rs) = set rs by (simp add: remdups-rev-def)


lemma remdups-rev-removeAll: remdups-rev (removeAll r rs) = removeAll r (remdups-rev
rs)
 by (simp add: remdups-filter remdups-rev-def removeAll-filter-not-eq rev-filter)
```

Faster code equations

```
fun remdups-rev-code :: 'a list ⇒ 'a list ⇒ 'a list where
  remdups-rev-code - [] = [] |
  remdups-rev-code ps (r#rs) = (if r ∈ set ps then remdups-rev-code ps rs else
r#remdups-rev-code (r#ps) rs)

lemma remdups-rev-code[code]: remdups-rev rs = remdups-rev-code [] rs
proof −
 { fix ps1 ps2 p and rs::'a list
   have set ps1 = set ps2 ⟹ remdups-rev-code ps1 rs = remdups-rev-code ps2 rs
     proof(induction rs arbitrary: ps1 ps2)
     case Nil thus ?case by simp
     next
     case (Cons r rs) show ?case
       apply(subst remdups-rev-code.simps)+
       apply(case-tac r ∈ set ps1)
        using Cons apply metis
```

```
      using Cons apply(simp)
      done
    qed
  } note remdups-rev-code-ps-seteq=this
  { fix ps1 ps2 p and rs::'a list
    have remdups-rev-code (ps1@ps2) rs = remdups-rev-code ps2 (filter (λr. r ∉
set ps1) rs)
    proof(induction rs arbitrary: ps1 ps2)
    case (Cons r rs)
      have remdups-rev-code (r # ps1 @ ps2) rs = remdups-rev-code (ps1 @ r
# ps2) rs
       by(rule remdups-rev-code-ps-seteq) simp
    with Cons.IH have remdups-rev-code (r # ps1 @ ps2) rs = remdups-rev-code
(r#ps2) [r←rs . r ∉ set ps1] by simp
      from this show ?case by(simp add: Cons)
    qed(simp add: remdups-rev-def)
   } note remdups-rev-code-ps-append=this
  { fix ps p and rs::'a list
    have remdups-rev-code (p # ps) rs = remdups-rev-code ps (removeAll p rs)
    by(simp add: remdups-rev-code-ps-append[of [p] ps rs, simplified] removeAll-filter-not-eq)
metis
  } note remdups-rev-code-ps-fst=this
  { fix ps p and rs::'a list
    have remdups-rev-code ps (removeAll p rs) = removeAll p (remdups-rev-code
ps rs)
    apply(induction rs arbitrary: ps)
     apply(simp-all)
    apply(safe)
     apply(simp-all)
    apply(simp add: remdups-rev-code-ps-fst removeAll-filter-not-eq)
    done
  } note remdups-rev-code-removeAll=this
  {fix ps
   have ∀ p ∈ set ps. p ∉ set rs ⟹ remdups-rev rs = remdups-rev-code ps rs
    apply(induction rs arbitrary: ps)
     apply(simp add: remdups-rev-def)
    apply(simp add: remdups-rev-fst remdups-rev-removeAll)
    apply safe
      apply(simp-all)
     apply(simp add: remdups-rev-code-ps-fst remdups-rev-code-removeAll)
     apply metis
    by blast
  }
  thus ?thesis by simp
qed


end
theory Ipassmt
```

imports *Common-Primitive-Syntax*
../Semantics-Ternary/Primitive-Normalization
Simple-Firewall.Iface
Simple-Firewall.IP-Addr-WordInterval-toString
Automatic-Refinement.Misc
begin
hide-const *Misc.uncurry*
hide-fact *Misc.uncurry-def*

A mapping from an interface to its assigned ip addresses in CIDR notation

type-synonym *'i ipassignment=iface* ⇀ (*'i word* × *nat*) *list*

31.1 Sanity checking for an *'i ipassignment*.

warning if interface map has wildcards

definition *ipassmt-sanity-nowildcards* :: *'i ipassignment* ⇒ *bool* **where**
ipassmt-sanity-nowildcards ipassmt ≡ ∀ *iface* ∈ *dom ipassmt*. ¬ *iface-is-wildcard iface*

Executable of the *'i ipassignment* is given as a list.

lemma[*code-unfold*]: *ipassmt-sanity-nowildcards* (*map-of ipassmt*) ⟷ (∀ *iface* ∈ *fst' set ipassmt*. ¬ *iface-is-wildcard iface*)
by(*simp add*: *ipassmt-sanity-nowildcards-def Map.dom-map-of-conv-image-fst*)

lemma *ipassmt-sanity-nowildcards-match-iface*:
ipassmt-sanity-nowildcards ipassmt ⟹
ipassmt (*Iface ifce2*) = *None* ⟹
ipassmt ifce = *Some a* ⟹
¬ *match-iface ifce ifce2*
unfolding *ipassmt-sanity-nowildcards-def*
by (*metis domIff iface.exhaust iface.sel option.distinct*(*1*) *iface-is-wildcard-def match-iface-case-nowildcard*)

definition *map-of-ipassmt* :: (*iface* × (*'i word* × *nat*) *list*) *list* ⇒ *iface* ⇀ (*'i word* × *nat*) *list* **where**
map-of-ipassmt ipassmt = (
if
distinct (*map fst ipassmt*) ∧ *ipassmt-sanity-nowildcards* (*map-of ipassmt*)
then
map-of ipassmt
else undefined ~~*(*~~*undefined ipassmt invalid, be distinct and don't have wildcard interfaces*~~*)*~~

some additional (optional) sanity checks

sanity check that there are no zone-spanning interfaces

definition *ipassmt-sanity-disjoint* :: *'i::len ipassignment* ⇒ *bool* **where**
ipassmt-sanity-disjoint ipassmt ≡ ∀ *i1* ∈ *dom ipassmt*. ∀ *i2* ∈ *dom ipassmt*. *i1* ≠ *i2* ⟶
ipcidr-union-set (*set* (*the* (*ipassmt i1*))) ∩ *ipcidr-union-set* (*set* (*the* (*ipassmt i2*))) = {}

lemma[*code-unfold*]: *ipassmt-sanity-disjoint* (*map-of ipassmt*) ⟷
(*let Is* = *fst*‘ *set ipassmt in*
(∀ *i1* ∈ *Is*. ∀ *i2* ∈ *Is*. *i1* ≠ *i2* ⟶ *wordinterval-empty* (*wordinterval-intersection* (*l2wi* (*map ipcidr-to-interval* (*the* ((*map-of ipassmt*) *i1*)))) (*l2wi* (*map ipcidr-to-interval* (*the* ((*map-of ipassmt*) *i2*)))))))
apply(*simp add*: *ipassmt-sanity-disjoint-def Map.dom-map-of-conv-image-fst*)
apply(*simp add*: *ipcidr-union-set-def*)
apply(*simp add*: *l2wi*)
apply(*simp add*: *ipcidr-to-interval-def*)
using *ipset-from-cidr-ipcidr-to-interval* **by** *blast*

Checking that the ipassmt covers the complete ipv4 address space.

definition *ipassmt-sanity-complete* :: (*iface* × (*'i::len word* × *nat*) *list*) *list* ⇒ *bool* **where**
ipassmt-sanity-complete ipassmt ≡ *distinct* (*map fst ipassmt*) ∧ (⋃(*ipcidr-union-set* ‘ *set* ‘ (*ran* (*map-of ipassmt*)))) = *UNIV*

lemma[*code*]: *ipassmt-sanity-complete ipassmt* ⟷ *distinct* (*map fst ipassmt*) ∧ (*let range* = *map snd ipassmt in*
wordinterval-eq (*wordinterval-Union* (*map* (*l2wi* ∘ (*map ipcidr-to-interval*)) *range*)) *wordinterval-UNIV*
)
apply(*cases distinct* (*map fst ipassmt*))
apply(*simp add*: *ipassmt-sanity-complete-def*)
apply(*simp add*: *Map.ran-distinct*)
apply(*simp add*: *wordinterval-eq-set-eq wordinterval-Union*)
apply(*simp add*: *l2wi*)
apply(*simp add*: *ipcidr-to-interval-def*)
apply(*simp add*: *ipcidr-union-set-def ipset-from-cidr-ipcidr-to-interval*; *fail*)
apply(*simp add*: *ipassmt-sanity-complete-def*)
done

value[*code*] *ipassmt-sanity-nowildcards* (*map-of* [(*Iface ''eth1.1017''*, [(*ipv4addr-of-dotdecimal* (*131*,*159*,*14*,*240*), *28*)])])

fun *collect-ifaces'* :: *'i::len common-primitive rule list* ⇒ *iface list* **where**
collect-ifaces' [] = [] |
collect-ifaces' ((*Rule m a*)#*rs*) = *filter* (λ*iface*. *iface* ≠ *ifaceAny*) (
(*map* (λ*x*. *case x of Pos i* ⇒ *i* | *Neg i* ⇒ *i*) (*fst* (*primitive-extractor* (*is-Iiface*, *iiface-sel*) *m*))) @
(*map* (λ*x*. *case x of Pos i* ⇒ *i* | *Neg i* ⇒ *i*) (*fst*

(*primitive-extractor* (*is-Oiface*, *oiface-sel*) *m*))) @ *collect-ifaces′ rs*)

definition *collect-ifaces* :: *′i*::*len common-primitive rule list* ⇒ *iface list* **where**
collect-ifaces rs ≡ *mergesort-remdups* (*collect-ifaces′ rs*)
lemma *set* (*collect-ifaces rs*) = *set* (*collect-ifaces′ rs*)
by(*simp add*: *collect-ifaces-def mergesort-remdups-correct*)

sanity check that all interfaces mentioned in the ruleset are also listed in the ipassmt. May fail for wildcard interfaces in the ruleset.

definition *ipassmt-sanity-defined* :: *′i*::*len common-primitive rule list* ⇒ *′i ipassignment* ⇒ *bool* **where**
ipassmt-sanity-defined rs ipassmt ≡ ∀ *iface* ∈ *set* (*collect-ifaces rs*). *iface* ∈ *dom ipassmt*

lemma[*code*]: *ipassmt-sanity-defined rs ipassmt* ⟷ (∀ *iface* ∈ *set* (*collect-ifaces rs*). *ipassmt iface* ≠ *None*)
by(*simp add*: *ipassmt-sanity-defined-def Map.domIff*)

lemma *ipassmt-sanity-defined* [
Rule (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*)) *24*))) (*Match* (*IIface* (*Iface ″eth1.1017″*)))) *action.Accept*,
Rule (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*)) *24*))) (*Match* (*IIface* (*ifaceAny*)))) *action.Accept*,
Rule MatchAny action.Drop]
(*map-of* [(*Iface ″eth1.1017″*, [(*ipv4addr-of-dotdecimal* (*131*,*159*,*14*,*240*), *28*)])]) **by** *eval*

definition *ipassmt-ignore-wildcard* :: *′i*::*len ipassignment* ⇒ *′i ipassignment* **where**
ipassmt-ignore-wildcard ipassmt ≡ λ*k*. *case ipassmt k of None* ⇒ *None*
| *Some ips* ⇒ *if ipcidr-union-set* (*set ips*) = *UNIV then None else Some ips*

lemma *ipassmt-ignore-wildcard-le*: *ipassmt-ignore-wildcard ipassmt* ⊆$_m$ *ipassmt*
apply(*simp add*: *ipassmt-ignore-wildcard-def map-le-def*)
apply(*clarify*)
apply(*simp split*: *option.split-asm if-split-asm*)
done

definition *ipassmt-ignore-wildcard-list*:: (*iface* × (*′i*::*len word* × *nat*) *list*) *list* ⇒ (*iface* × (*′i word* × *nat*) *list*) *list* **where**
ipassmt-ignore-wildcard-list ipassmt = *filter* (λ(-,*ips*). ¬ *wordinterval-eq* (*l2wi* (*map ipcidr-to-interval ips*)) *wordinterval-UNIV*) *ipassmt*

lemma *distinct* (*map fst ipassmt*) ⟹
map-of (*ipassmt-ignore-wildcard-list ipassmt*) = *ipassmt-ignore-wildcard* (*map-of*

ipassmt)
apply(*simp add*: *ipassmt-ignore-wildcard-list-def ipassmt-ignore-wildcard-def*)
apply(*simp add*: *wordinterval-eq-set-eq*)
apply(*simp add*: *l2wi*)
apply(*simp add*: *ipcidr-to-interval-def*)
apply(*simp add*: *fun-eq-iff*)
apply(*clarify*)
apply(*induction ipassmt*)
apply(*simp*; *fail*)
apply(*simp*)
apply(*simp split*:*option.split option.split-asm*)
apply(*simp add*: *ipcidr-union-set-def ipset-from-cidr-ipcidr-to-interval*)
apply(*simp add*: *case-prod-unfold*)
by *blast*

Debug algorithm with human-readable output

definition *debug-ipassmt-generic*
:: ('*i*::*len wordinterval* ⇒ *string*) ⇒
(*iface* × ('*i word* × *nat*) *list*) *list* ⇒ '*i common-primitive rule list* ⇒ *string list* **where**
debug-ipassmt-generic toStr ipassmt rs ≡ *let ifaces* = (*map fst ipassmt*) *in* [
''*distinct*: '' @ (*if distinct ifaces then* ''*passed*'' *else* ''*FAIL*!'')
, ''*ipassmt-sanity-nowildcards*: '' @
(*if ipassmt-sanity-nowildcards* (*map-of ipassmt*)
then ''*passed*'' *else* ''*fail*: ''@*list-toString iface-sel* (*filter iface-is-wildcard ifaces*))
, ''*ipassmt-sanity-defined* (*interfaces defined in the ruleset are also in ipassmt*): '' @
(*if ipassmt-sanity-defined rs* (*map-of ipassmt*)
then ''*passed*'' *else* ''*fail*: ''@*list-toString iface-sel* [*i* ← (*collect-ifaces rs*). *i* ∉ *set ifaces*])
, ''*ipassmt-sanity-disjoint* (*no zone−spanning interfaces*): '' @
(*if ipassmt-sanity-disjoint* (*map-of ipassmt*)
then ''*passed*'' *else* ''*fail*: ''@*list-toString* (λ(*i1*,*i2*). ''('' @ *iface-sel i1* @ '','' @ *iface-sel i2* @ '')'')
[(*i1*,*i2*) ← *List.product ifaces ifaces*. *i1* ≠ *i2* ∧
¬ *wordinterval-empty* (*wordinterval-intersection*
(*l2wi* (*map ipcidr-to-interval* (*the* ((*map-of ipassmt*) *i1*))))
(*l2wi* (*map ipcidr-to-interval* (*the* ((*map-of ipassmt*) *i2*)))))
])
, ''*ipassmt-sanity-disjoint excluding UNIV interfaces*: '' @
(*let ipassmt* = *ipassmt-ignore-wildcard-list ipassmt*;
ifaces = (*map fst ipassmt*)
in
(*if ipassmt-sanity-disjoint* (*map-of ipassmt*)
then ''*passed*'' *else* ''*fail*: ''@*list-toString* (λ(*i1*,*i2*). ''('' @ *iface-sel i1* @ '','' @ *iface-sel i2* @ '')'')

```
            [(i1,i2) ← List.product ifaces ifaces. i1 ≠ i2 ∧
             ¬ wordinterval-empty (wordinterval-intersection
                                   (l2wi (map ipcidr-to-interval (the ((map-of ipassmt)
i1))))
                                   (l2wi (map ipcidr-to-interval (the ((map-of ipassmt)
i2)))))
          ]))
       , ''ipassmt-sanity-complete: '' @
          (if ipassmt-sanity-complete ipassmt
           then ''passed''
           else ''the following is not covered: '' @
              toStr (wordinterval-setminus wordinterval-UNIV (wordinterval-Union
(map (l2wi ∘ (map ipcidr-to-interval)) (map snd ipassmt)))))
       , ''ipassmt-sanity-complete excluding UNIV interfaces: '' @
          (let ipassmt = ipassmt-ignore-wildcard-list ipassmt
           in
          (if ipassmt-sanity-complete ipassmt
           then ''passed''
           else ''the following is not covered: '' @
              toStr (wordinterval-setminus wordinterval-UNIV (wordinterval-Union
(map (l2wi ∘ (map ipcidr-to-interval)) (map snd ipassmt))))))
      ]

 definition debug-ipassmt-ipv4 ≡ debug-ipassmt-generic ipv4addr-wordinterval-toString
 definition debug-ipassmt-ipv6 ≡ debug-ipassmt-generic ipv6addr-wordinterval-toString


  lemma dom-ipassmt-ignore-wildcard:
   i∈dom (ipassmt-ignore-wildcard ipassmt) ⟷ i ∈ dom ipassmt ∧ ipcidr-union-set
(set (the (ipassmt i))) ≠ UNIV
    apply(simp add: ipassmt-ignore-wildcard-def)
    apply(rule)
     apply(clarify)
     apply(simp split: option.split-asm if-split-asm)
     apply blast
    apply(clarify)
    apply(simp)
    done

  lemma ipassmt-ignore-wildcard-the:
   ipassmt i = Some ips ⟹ ipcidr-union-set (set ips) ≠ UNIV ⟹ (the (ipassmt-ignore-wildcard
ipassmt i)) = ips
    ipassmt-ignore-wildcard ipassmt i = Some ips ⟹ the (ipassmt i) = ips
    ipassmt-ignore-wildcard ipassmt i = Some ips ⟹ ipcidr-union-set (set ips) ≠
UNIV
   by (simp-all add: ipassmt-ignore-wildcard-def split: option.split-asm if-split-asm)


  lemma ipassmt-sanity-disjoint-ignore-wildcards:
```

ipassmt-sanity-disjoint (*ipassmt-ignore-wildcard ipassmt*) ⟷
(∀ *i1*∈*dom ipassmt*.
∀ *i2*∈*dom ipassmt*.
ipcidr-union-set (*set* (*the* (*ipassmt i1*))) ≠ *UNIV* ∧
ipcidr-union-set (*set* (*the* (*ipassmt i2*))) ≠ *UNIV* ∧
i1 ≠ *i2*
⟶ *ipcidr-union-set* (*set* (*the* (*ipassmt i1*))) ∩ *ipcidr-union-set* (*set* (*the* (*ipassmt i2*))) = {})
apply(*simp add*: *ipassmt-sanity-disjoint-def*)
apply(*rule*)
apply(*clarify*)
apply(*simp*)
subgoal for *i1 i2 ips1 ips2*
apply(*erule-tac x*=*i1* **in** *ballE*)
prefer *2*
using *dom-ipassmt-ignore-wildcard* **apply** (*metis domI option.sel*)
apply(*erule-tac x*=*i2* **in** *ballE*)
prefer *2*
using *dom-ipassmt-ignore-wildcard* **apply** (*metis domI domIff option.sel*)
by(*simp add*: *ipassmt-ignore-wildcard-the*; *fail*)
apply(*clarify*)
apply(*simp*)
subgoal for *i1 i2 ips1 ips2*
apply(*erule-tac x*=*i1* **in** *ballE*)
prefer *2*
using *dom-ipassmt-ignore-wildcard* **apply** *auto*[*1*]
apply(*erule-tac x*=*i2* **in** *ballE*)
prefer *2*
using *dom-ipassmt-ignore-wildcard* **apply** *auto*[*1*]
by(*simp add*: *ipassmt-ignore-wildcard-the*)
done

Confusing names: *ipassmt-sanity-nowildcards* refers to wildcard interfaces. *ipassmt-ignore-wildcard* refers to the UNIV ip range.

lemma *ipassmt-sanity-nowildcards-ignore-wildcardD*:
ipassmt-sanity-nowildcards ipassmt ⟹ *ipassmt-sanity-nowildcards* (*ipassmt-ignore-wildcard ipassmt*)
by (*simp add*: *dom-ipassmt-ignore-wildcard ipassmt-sanity-nowildcards-def*)

lemma *ipassmt-disjoint-nonempty-inj*:
assumes *ipassmt-disjoint*: *ipassmt-sanity-disjoint ipassmt*
and *ifce*: *ipassmt ifce* = *Some i-ips*
and *a*: *ipcidr-union-set* (*set i-ips*) ≠ {}
and *k*: *ipassmt k* = *Some i-ips*
shows *k* = *ifce*
proof(*rule ccontr*)
assume *k* ≠ *ifce*
with *ifce k ipassmt-disjoint* **have** *ipcidr-union-set* (*set* (*the* (*ipassmt k*))) ∩

ipcidr-union-set (*set* (*the* (*ipassmt ifce*))) = {}
unfolding *ipassmt-sanity-disjoint-def* **by** *fastforce*
thus *False* **using** *a ifce k* **by** *auto*
qed

lemma *ipassmt-ignore-wildcard-None-Some*:
ipassmt-ignore-wildcard ipassmt ifce = *None* ⟹ *ipassmt ifce* = *Some ips* ⟹ *ipcidr-union-set* (*set ips*) = *UNIV*
by (*metis domI domIff dom-ipassmt-ignore-wildcard option.sel*)

lemma *ipassmt-disjoint-ignore-wildcard-nonempty-inj*:
assumes *ipassmt-disjoint*: *ipassmt-sanity-disjoint* (*ipassmt-ignore-wildcard ipassmt*)
and *ifce*: *ipassmt ifce* = *Some i-ips*
and *a*: *ipcidr-union-set* (*set i-ips*) ≠ {}
and *k*: (*ipassmt-ignore-wildcard ipassmt*) *k* = *Some i-ips*
shows *k* = *ifce*
proof(*rule ccontr*)
assume *k* ≠ *ifce*
show *False*
proof(*cases* (*ipassmt-ignore-wildcard ipassmt*) *ifce*)
case (*Some i-ips'*)
hence *i-ips'* = *i-ips* **using** *ifce ipassmt-ignore-wildcard-the*(*2*) **by** *fastforce*
hence (*ipassmt-ignore-wildcard ipassmt*) *k* = *Some i-ips* **using** *Some ifce ipassmt-ignore-wildcard-def k* **by** *auto*
thus *False* **using** *Some* ‹*i-ips'* = *i-ips*› ‹*k* ≠ *ifce*› *a ipassmt-disjoint ipassmt-disjoint-nonempty-inj* **by** *blast*
next
case *None*
with *ipassmt-ignore-wildcard-None-Some* **have** *ipcidr-union-set* (*set i-ips*) = *UNIV* **using** *ifce* **by** *auto*
thus *False* **using** *ipassmt-ignore-wildcard-the*(*3*) *k* **by** *blast*
qed
qed

lemma *ipassmt-disjoint-inj-k*:
assumes *ipassmt-disjoint*: *ipassmt-sanity-disjoint ipassmt*
and *ifce*: *ipassmt ifce* = *Some ips*
and *k*: *ipassmt k* = *Some ips'*
and *a*: *p* ∈ *ipcidr-union-set* (*set ips*)
and *b*: *p* ∈ *ipcidr-union-set* (*set ips'*)
shows *k* = *ifce*
proof(*rule ccontr*)
assume *k* ≠ *ifce*
with *ipassmt-disjoint* **have**
ipcidr-union-set (*set* (*the* (*ipassmt k*))) ∩ *ipcidr-union-set* (*set* (*the* (*ipassmt ifce*))) = {}

unfolding *ipassmt-sanity-disjoint-def* **using** *ifce k* **by** *blast*
hence *ipcidr-union-set* (*set ips'*) ∩ *ipcidr-union-set* (*set ips*) = {} **by**(*simp add*: *k ifce*)
thus *False* **using** *a b* **by** *blast*
qed

lemma *ipassmt-disjoint-matcheq-iifce-srcip*:
assumes *ipassmt-nowild*: *ipassmt-sanity-nowildcards ipassmt*
and *ipassmt-disjoint*: *ipassmt-sanity-disjoint ipassmt*
and *ifce*: *ipassmt ifce* = *Some i-ips*
and *p-ifce*: *ipassmt* (*Iface* (*p-iiface p*)) = *Some p-ips* ∧ *p-src p* ∈ *ipcidr-union-set* (*set p-ips*)
shows *match-iface ifce* (*p-iiface p*) ⟷ *p-src p* ∈ *ipcidr-union-set* (*set i-ips*)
proof
assume *match-iface ifce* (*p-iiface p*)
thus *p-src p* ∈ *ipcidr-union-set* (*set i-ips*)
apply(*cases ifce* = *Iface* (*p-iiface p*))
using *ifce p-ifce* **apply** *force*
by (*metis domI iface.sel iface-is-wildcard-def ifce ipassmt-nowild ipassmt-sanity-nowildcards-def match-iface.elims*(2) *match-iface-case-nowildcard*)
next
assume *a*: *p-src p* ∈ *ipcidr-union-set* (*set i-ips*)
— basically, we need to reverse the map *ipassmt*

from *ipassmt-disjoint-nonempty-inj*[*OF ipassmt-disjoint ifce*] *a* **have** *ipassmt-inj*: ∀ *k*. *ipassmt k* = *Some i-ips* ⟶ *k* = *ifce* **by** *blast*

from *ipassmt-disjoint-inj-k*[*OF ipassmt-disjoint ifce - a*] **have** *ipassmt-inj-k*: ⋀*k ips'*. *ipassmt k* = *Some ips'* ⟹ *p-src p* ∈ *ipcidr-union-set* (*set ips'*) ⟹ *k* = *ifce* **by** *simp*

have *ipassmt-inj-p*: ∀ *ips'*. *p-src p* ∈ *ipcidr-union-set* (*set ips'*) ∧ (∃ *k*. *ipassmt k* = *Some ips'*) ⟶ *ips'* = *i-ips*
apply(*clarify*)
apply(*rename-tac ips' k*)
apply(*subgoal-tac k* = *ifce*)
using *ifce* **apply** *simp*
using *ipassmt-inj-k* **by** *simp*

from *p-ifce* **have** (*Iface* (*p-iiface p*)) = *ifce* **using** *ipassmt-inj-p ipassmt-inj* **by** *blast*

thus *match-iface ifce* (*p-iiface p*) **using** *match-iface-refl* **by** *blast*
qed

definition *ipassmt-generic-ipv4* :: (*iface* × (*32 word* × *nat*) *list*) *list* **where**
ipassmt-generic-ipv4 = [(*Iface* ''*lo*'', [(*ipv4addr-of-dotdecimal* (*127*,*0*,*0*,*0*),*8*)])]

definition *ipassmt-generic-ipv6* :: (*iface* × (*128 word* × *nat*) *list*) *list* **where**
ipassmt-generic-ipv6 = [(*Iface* ''*lo*'', [(*1*,*128*)])]

31.2 IP Assignment difference

Compare two ipassmts. Returns a list of tuples First entry of the tuple: things which are in the left ipassmt but not in the right. Second entry of the tupls: things which are in the right ipassmt but not in the left.

definition *ipassmt-diff*
:: (*iface* × (*'i*::*len word* × *nat*) *list*) *list* ⇒ (*iface* × (*'i*::*len word* × *nat*) *list*) *list*
⇒ (*iface* × (*'i word* × *nat*) *list* × (*'i word* × *nat*) *list*) *list*
where
ipassmt-diff a b ≡ *let*
t = λ*s*. (*case s of None* ⇒ *Empty-WordInterval*
| *Some s* ⇒ *wordinterval-Union* (*map ipcidr-tuple-to-wordinterval s*));
k = λ*x y d*. *cidr-split* (*wordinterval-setminus* (*t* (*map-of x d*)) (*t* (*map-of y d*)))
in
[(*d*, (*k a b d*, *k b a d*)). *d* ← *remdups* (*map fst* (*a* @ *b*))]

If an interface is defined in both ipassignments and there is no difference then the two ipassignements describe the same IP range for this interface.

lemma *ipassmt-diff-ifce-equal*: (*ifce*, [], []) ∈ *set* (*ipassmt-diff ipassmt1 ipassmt2*) ⟹
ifce ∈ *dom* (*map-of ipassmt1*) ⟹ *ifce* ∈ *dom* (*map-of ipassmt2*) ⟹
ipcidr-union-set (*set* (*the* ((*map-of ipassmt1*) *ifce*))) =
ipcidr-union-set (*set* (*the* ((*map-of ipassmt2*) *ifce*)))
proof −
have *cidr-empty*: [] = *cidr-split r* ⟹ *wordinterval-to-set r* = {} **for** *r* :: *'a wordinterval*
apply(*subst cidr-split-prefix*[*symmetric*])
by(*simp*)
show (*ifce*, [], []) ∈ *set* (*ipassmt-diff ipassmt1 ipassmt2*) ⟹
ifce ∈ *dom* (*map-of ipassmt1*) ⟹ *ifce* ∈ *dom* (*map-of ipassmt2*) ⟹
ipcidr-union-set (*set* (*the* ((*map-of ipassmt1*) *ifce*))) =
ipcidr-union-set (*set* (*the* ((*map-of ipassmt2*) *ifce*)))
apply(*simp add*: *ipassmt-diff-def Let-def ipcidr-union-set-uncurry*)
apply(*simp add*: *Set.image-iff*)
apply(*elim conjE*)
apply(*drule cidr-empty*)+
apply(*simp*)
apply(*simp add*: *domIff*)
apply(*elim exE*)

apply(*simp add*: *wordinterval-Union wordinterval-to-set-ipcidr-tuple-to-wordinterval-uncurry*)
done
qed

lemma *ipcidr-union-cidr-split*[*simp*]: *ipcidr-union-set* (*set* (*cidr-split a*)) = *wordinterval-to-set a*
by(*simp add*: *ipcidr-union-set-uncurry cidr-split-prefix*)

lemma
defines *assmt as ifce* ≡ *ipcidr-union-set* (*set* (*the* ((*map-of as ifce*))))
assumes *diffs*: (*ifce*, *d1*, *d2*) ∈ *set* (*ipassmt-diff ipassmt1 ipassmt2*)
and *doms*: *ifce* ∈ *dom* (*map-of ipassmt1*) *ifce* ∈ *dom* (*map-of ipassmt2*)
shows *ipcidr-union-set* (*set d1*) = *assmt ipassmt1 ifce* − *assmt ipassmt2 ifce*
ipcidr-union-set (*set d2*) = *assmt ipassmt2 ifce* − *assmt ipassmt1 ifce*
using *assms* **by** (*clarsimp simp add*: *ipassmt-diff-def Let-def assmt-def wordinterval-Union*; *simp add*: *ipcidr-union-set-uncurry uncurry-def wordinterval-to-set-ipcidr-tuple-to-wordinterval-unc

Explanation for interface *Iface ''a''*: Left ipassmt: The IP range 4/30 contains the addresses 4,5,6,7 Diff: right ipassmt contains 6/32, so 4,5,7 is only in the left ipassmt. IP addresses 4,5 correspond to subnet 4/30.

lemma *ipassmt-diff* (*ipassmt-generic-ipv4* @ [(*Iface ''a''*, [(*4*,*30*)])])
(*ipassmt-generic-ipv4* @ [(*Iface ''a''*, [(*6*,*32*), (*0*,*30*)]), (*Iface ''b''*, [(*42*,*32*)])]) =
[(*Iface ''lo''*, [], []),
(*Iface ''a''*, [(*4*, *31*),(*7*, *32*)],
[(*0*, *30*)]
),
(*Iface ''b''*, [], [(*42*, *32*)])] **by** *eval*

end
theory *No-Spoof*
imports *Common-Primitive-Lemmas*
Ipassmt
begin

32 No Spoofing

assumes: *simple-ruleset*

32.1 Spoofing Protection

No spoofing means: Every packet that is (potentially) allowed by the firewall and comes from an interface *iface* must have a Source IP Address in the assigned range *iface*.

"potentially allowed" means we use the upper closure. The definition states: For all interfaces which are configured, every packet that comes from this

interface and is allowed by the firewall must be in the IP range of that interface.

We add *'pkt-ext itself* as a parameter to have the type of a generic, extensible packet in the definition.

definition *no-spoofing* :: *'pkt-ext itself* ⇒ *'i::len ipassignment* ⇒ *'i::len common-primitive rule list* ⇒ *bool* **where**
no-spoofing TYPE('pkt-ext) ipassmt rs ≡ ∀ *iface* ∈ *dom ipassmt*. ∀ *p* :: (*'i*,*'pkt-ext*) *tagged-packet-scheme*.
((*common-matcher*, *in-doubt-allow*),*p*⟦*p-iiface*:=*iface-sel iface*⟧⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow*) ⟶
p-src p ∈ (*ipcidr-union-set* (*set* (*the* (*ipassmt iface*))))

This is how it looks like for an IPv4 simple packet: We add *unit* because a *32 tagged-packet* does not have any additional fields.

lemma *no-spoofing TYPE*(*unit*) *ipassmt rs* ⟷
(∀ *iface* ∈ *dom ipassmt*. ∀ *p* :: *32 tagged-packet*.
((*common-matcher*, *in-doubt-allow*),*p*⟦*p-iiface*:=*iface-sel iface*⟧⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow*)
⟶ *p-src p* ∈ (*ipcidr-union-set* (*set* (*the* (*ipassmt iface*)))))
unfolding *no-spoofing-def* **by** *blast*

The definition is sound (if that can be said about a definition): if *no-spoofing* certifies your ruleset, then your ruleset prohibits spoofing. The definition may not be complete: *no-spoofing* may return *False* even though your ruleset prevents spoofing (should only occur if some strange and unknown primitives occur)

Technical note: The definition can can be thought of as protection from OUTGOING spoofing. OUTGOING means: I define my interfaces and their IP addresses. For all interfaces, only the assigned IP addresses may pass the firewall. This definition is simple for e.g. local sub-networks. Example: [*Iface ''eth0''* ↦ {(*ipv4addr-of-dotdecimal* (*192*, *168*, *0*, *0*), *24*::*'a*)}]
If I want spoofing protection from the Internet, I need to specify the range of the Internet IP addresses. Example: [*Iface ''evil-internet''* ↦ {*everything-that-does-not-belong-to-me*}]
This is also a good opportunity to exclude the private IP space, link local, and probably multicast space. See *all-but-those-ips* to easily specify these ranges.
See examples below. Check Example 3 why it can be thought of as OUTGOING spoofing.

context
begin

The set of any ip addresses which may match for a fixed *iface* (overapproximation)

private definition *get-exists-matching-src-ips* :: *iface* ⇒ *'i*::*len common-primitive match-expr* ⇒ *'i word set* **where**
get-exists-matching-src-ips iface m ≡ *let* (*i-matches*, -) = (*primitive-extractor* (*is-Iface*, *iiface-sel*) *m*) *in*
if (∀ *is* ∈ *set i-matches*. (*case is of Pos i* ⇒ *match-iface i* (*iface-sel iface*)
| *Neg i* ⇒ ¬ *match-iface i* (*iface-sel iface*)))
then
(*let* (*ip-matches*, -) = (*primitive-extractor* (*is-Src*, *src-sel*) *m*) *in*
if ip-matches = []
then
UNIV
else
⋂ *ips* ∈ *set* (*ip-matches*). (*case ips of Pos ip* ⇒ *ipt-iprange-to-set ip* | *Neg ip* ⇒ − *ipt-iprange-to-set ip*))
else
{}

lemma *primitive-extractor* (*is-Src*, *src-sel*)
(*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (*0*::*ipv4addr*) *30*))) (*Match* (*IIface* (*Iface ''eth0''*)))) =
([*Pos* (*IpAddrNetmask 0 30*)], *MatchAnd MatchAny* (*Match* (*IIface* (*Iface ''eth0''*)))) **by** *eval*

private lemma *get-exists-matching-src-ips-subset*:
assumes *normalized-nnf-match m*
shows {*ip*. (∃ *p* :: (*'i*::*len*, *'a*) *tagged-packet-scheme*. *matches* (*common-matcher*, *in-doubt-allow*) *m a* (*p*⦇*p-iiface*:= *iface-sel iface*, *p-src*:= *ip*⦈))} ⊆
get-exists-matching-src-ips iface m
proof −
let *?γ*=(*common-matcher*, *in-doubt-allow*)

{ fix *ip-matches rest src-ip i-matches rest2* **and** *p* :: (*'i*, *'a*) *tagged-packet-scheme*
assume *a1*: *primitive-extractor* (*is-Src*, *src-sel*) *m* = (*ip-matches*, *rest*)
and *a2*: *matches ?γ m a* (*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *src-ip*⦈)
let *?p*=(*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *src-ip*⦈)

from *primitive-extractor-negation-type-matching1*[*OF wf-disc-sel-common-primitive*(*3*) *assms a1 a2*]
match-simplematcher-SrcDst[**where** *p* = *?p*] *match-simplematcher-SrcDst-not*[**where** *p*=*?p*]
have *ip-matches*: (∀ *ip*∈*set* (*getPos ip-matches*). *p-src ?p* ∈ *ipt-iprange-to-set ip*) ∧
(∀ *ip*∈*set* (*getNeg ip-matches*). *p-src ?p* ∈ − *ipt-iprange-to-set ip*) **by** *simp*
from *ip-matches* **have** ∀ *x* ∈ *set ip-matches*. *src-ip* ∈ (*case x of Pos x* ⇒ *ipt-iprange-to-set x* | *Neg ip* ⇒ − *ipt-iprange-to-set ip*)
apply(*simp*)

```
      apply(simp  split: negation-type.split)
      apply(safe)
       using NegPos-set apply fast+
     done
   } note 1=this

   { fix ip-matches rest src-ip i-matches rest2 and p :: ('i, 'a) tagged-packet-scheme
     assume a1: primitive-extractor (is-Iiface, iiface-sel) m = (i-matches, rest2)
       and a2: matches ?γ m a (p(|p-iiface := iface-sel iface, p-src := src-ip|))
     let ?p=(p(|p-iiface := iface-sel iface, p-src := src-ip|))

    from primitive-extractor-negation-type-matching1[OF wf-disc-sel-common-primitive(5)
assms a1 a2]
       primitive-matcher-generic.Iface-single[OF primitive-matcher-generic-common-matcher,
where p = ?p]
       primitive-matcher-generic.Iface-single-not[OF primitive-matcher-generic-common-matcher,
where p = ?p]
     have iface-matches: (∀ i∈set (getPos i-matches). match-iface i (p-iiface ?p))
∧
                         (∀ i∈set (getNeg i-matches). ¬ match-iface i (p-iiface ?p))
by simp
     hence 2: (∀ x∈set i-matches. case x of Pos i ⇒ match-iface i (iface-sel iface)
| Neg i ⇒ ¬ match-iface i (iface-sel iface))
       apply(simp add: split: negation-type.split)
       apply(safe)
       using NegPos-set apply fast+
     done

   } note 2=this

   from 1 2 show ?thesis
     unfolding get-exists-matching-src-ips-def
     by(clarsimp)
 qed


 lemma common-primitive-not-has-primitive-expand:
      ¬ has-primitive (m::'i::len common-primitive match-expr) ⟷
       ¬ has-disc is-Dst m ∧
       ¬ has-disc is-Src m ∧
       ¬ has-disc is-Iiface m ∧
       ¬ has-disc is-Oiface m ∧
       ¬ has-disc is-Prot m ∧
       ¬ has-disc is-Src-Ports m ∧
       ¬ has-disc is-Dst-Ports m ∧
       ¬ has-disc is-MultiportPorts m ∧
       ¬ has-disc is-L4-Flags m ∧
       ¬ has-disc is-CT-State m ∧
       ¬ has-disc is-Extra m
```

apply(*induction m*)
apply(*simp-all*)
apply(*rename-tac x*, *case-tac x*, *simp-all*)
by *blast*

lemma ¬ *has-primitive m* ∧ *matcheq-matchAny m* ⟷ (*if* ¬ *has-primitive m then matcheq-matchAny m else False*)
by *simp*

The set of ip addresses which definitely match for a fixed *iface* (underapproximation)

private definition *get-all-matching-src-ips* :: *iface* ⇒ *'i*::*len common-primitive match-expr* ⇒ *'i word set* **where**
get-all-matching-src-ips iface m ≡ *let* (*i-matches*, *rest1*) = (*primitive-extractor* (*is-Iiface*, *iiface-sel*) *m*) *in*
if (∀ *is* ∈ *set i-matches*. (*case is of Pos i* ⇒ *match-iface i* (*iface-sel iface*)
| *Neg i* ⇒ ¬ *match-iface i* (*iface-sel iface*)))
then
(*let* (*ip-matches*, *rest2*) = (*primitive-extractor* (*is-Src*, *src-sel*) *rest1*) *in*
if ¬ *has-primitive rest2* ∧ *matcheq-matchAny rest2*
then
if ip-matches = []
then
UNIV
else
⋂ *ips* ∈ *set* (*ip-matches*). (*case ips of Pos ip* ⇒ *ipt-iprange-to-set ip* | *Neg ip* ⇒ − *ipt-iprange-to-set ip*)
else
{})
else
{}

private lemma *get-all-matching-src-ips*:
assumes *normalized-nnf-match m*
shows *get-all-matching-src-ips iface m* ⊆
{*ip*. (∀ *p*::(*'i*::*len*, *'a*) *tagged-packet-scheme*. *matches* (*common-matcher*, *in-doubt-allow*) *m a* (*p*⦇*p-iiface*:= *iface-sel iface*, *p-src*:= *ip*⦈))}
proof
fix *ip*
assume *a*: *ip* ∈ *get-all-matching-src-ips iface m*
obtain *i-matches rest1* **where** *select1*: *primitive-extractor* (*is-Iiface*, *iiface-sel*) *m* = (*i-matches*, *rest1*) **by** *fastforce*
show *ip* ∈ {*ip*. ∀ *p* :: (*'i*, *'a*) *tagged-packet-scheme*. *matches* (*common-matcher*,

```
in-doubt-allow) m a (p⦇p-iiface := iface-sel iface, p-src := ip⦈)}
    proof(cases ∀ is ∈ set i-matches. (case is of Pos i ⇒ match-iface i (iface-sel
iface)
                                        | Neg i ⇒ ¬match-iface i (iface-sel iface)))
   case False
    have get-all-matching-src-ips iface m = {}
     unfolding get-all-matching-src-ips-def
     using select1 False by auto
    with a show ?thesis by simp
   next
   case True
     let ?γ=(common-matcher, in-doubt-allow) :: ('i::len common-primitive, ('i,
'a) tagged-packet-scheme) match-tac
    let ?p=λp::('i, 'a) tagged-packet-scheme. p⦇p-iiface := iface-sel iface, p-src :=
ip⦈
     obtain ip-matches rest2 where select2: primitive-extractor (is-Src, src-sel)
rest1 = (ip-matches, rest2) by fastforce

    let ?noDisc=¬ has-primitive rest2

    have get-all-matching-src-ips-caseTrue: get-all-matching-src-ips iface m =
          (if ?noDisc ∧ matcheq-matchAny rest2
           then if ip-matches = []
                then UNIV
             else ⋂((case-negation-type ipt-iprange-to-set (λip. − ipt-iprange-to-set
ip) ` (set ip-matches)))
           else {})
    unfolding get-all-matching-src-ips-def
    by(simp add: True select1 select2)

   from True have (∀ m∈set (getPos i-matches). matches ?γ (Match (IIface m))
a (?p p)) ∧
                        (∀ m∈set (getNeg i-matches). matches ?γ (MatchNot (Match
(IIface m))) a (?p p))
     for p :: ('i, 'a) tagged-packet-scheme
      by(simp add: negation-type-forall-split
       primitive-matcher-generic.Iface-single[OF primitive-matcher-generic-common-matcher]
       primitive-matcher-generic.Iface-single-not[OF primitive-matcher-generic-common-matcher])
     hence matches-iface: matches ?γ (alist-and (NegPos-map IIface i-matches))
a (?p p)
      for p :: ('i,'a) tagged-packet-scheme
      by(simp add: matches-alist-and NegPos-map-simps)

    show ?thesis
    proof(cases ?noDisc ∧ matcheq-matchAny rest2)
    case False
     assume F: ¬ (?noDisc ∧ matcheq-matchAny rest2)
      with get-all-matching-src-ips-caseTrue have get-all-matching-src-ips iface
m = {} by presburger
```

with *a* **have** *False* **by** *simp*
thus *?thesis* **..**
next
case *True*
assume *F*: *?noDisc* ∧ *matcheq-matchAny rest2*
with *get-all-matching-src-ips-caseTrue* **have** *get-all-matching-src-ips iface m* =
(*if ip-matches* = []
then UNIV
else ⋂((*case-negation-type ipt-iprange-to-set* (λ*ip*. − *ipt-iprange-to-set ip*) ‘ (*set ip-matches*)))) **by** *presburger*

from *primitive-extractor-correct*[*OF assms wf-disc-sel-common-primitive*(5) *select1*] **have**
select1-matches: *matches ?γ* (*alist-and* (*NegPos-map IIface i-matches*)) *a p* ∧ *matches ?γ rest1 a p* ⟷ *matches ?γ m a p*
and *normalized1*: *normalized-nnf-match rest1* **for** *p* :: (′*i*,′*a*) *tagged-packet-scheme*
apply −
apply *fast+*
done
from *select1-matches matches-iface* **have**
rest1-matches: *matches ?γ rest1 a* (*?p p*) ⟷ *matches ?γ m a* (*?p p*) **for** *p* :: (′*i*, ′*a*) *tagged-packet-scheme* **by** *blast*

from *primitive-extractor-correct*[*OF normalized1 wf-disc-sel-common-primitive*(3) *select2*] **have**
select2-matches: *matches ?γ* (*alist-and* (*NegPos-map Src ip-matches*)) *a p* ∧ *matches ?γ rest2 a p* ⟷
matches ?γ rest1 a p **for** *p* :: (′*i*, ′*a*) *tagged-packet-scheme*
by *fast*
with *F matcheq-matchAny* **have** *matches ?γ rest2 a p* **for** *p* :: (′*i*, ′*a*) *tagged-packet-scheme* **by** *metis*
with *select2-matches rest1-matches* **have** *ip-src-matches*:
matches ?γ (*alist-and* (*NegPos-map Src ip-matches*)) *a* (*?p p*) ⟷ *matches ?γ m a* (*?p p*)
for *p* :: (′*i*, ′*a*) *tagged-packet-scheme* **by** *simp*

have *case-nil*: ⋀*p*. *ip-matches* = [] ⟹ *matches ?γ* (*alist-and* (*NegPos-map Src ip-matches*)) *a p*
by(*simp add*: *bunch-of-lemmata-about-matches*)

have *case-list*: ⋀*p*. ∀ *x*∈*set ip-matches*. (*case x of Pos i* ⇒ *ip* ∈ *ipt-iprange-to-set i*
| *Neg i* ⇒ *ip* ∈ − *ipt-iprange-to-set i*) ⟹
matches ?γ (*alist-and* (*NegPos-map Src ip-matches*)) *a* (*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *ip*⦈)
apply(*simp add*: *matches-alist-and NegPos-map-simps*)
apply(*simp add*: *negation-type-forall-split match-simplematcher-SrcDst-not*

match-simplematcher-SrcDst)
done

from *a* **show** *ip* ∈ {*ip*. ∀ *p* :: ('*i*, '*a*) *tagged-packet-scheme*. *matches* (*common-matcher*, *in-doubt-allow*) *m a* (*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *ip*⦈)}
unfolding *get-all-matching-src-ips-caseTrue*
proof(*clarsimp split*: *if-split-asm*)
fix *p* :: ('*i*, '*a*) *tagged-packet-scheme*
assume *ip-matches* = []
with *case-nil* **have** *matches* ?γ (*alist-and* (*NegPos-map Src ip-matches*)) *a* (?*p p*) **by** *simp*
with *ip-src-matches* **show** *matches* ?γ *m a* (?*p p*) **by** *simp*
next
fix *p* :: ('*i*, '*a*) *tagged-packet-scheme*
assume ∀ *x*∈*set ip-matches*. *ip* ∈ (*case x of Pos x* ⇒ *ipt-iprange-to-set x* | *Neg ip* ⇒ − *ipt-iprange-to-set ip*)
hence ∀ *x*∈*set ip-matches*. *case x of Pos i* ⇒ *ip* ∈ *ipt-iprange-to-set i* | *Neg i* ⇒ *ip* ∈ − *ipt-iprange-to-set i*
by(*simp-all split*: *negation-type.split negation-type.split-asm*)
with *case-list* **have** *matches* ?γ (*alist-and* (*NegPos-map Src ip-matches*)) *a* (?*p p*) **.**
with *ip-src-matches* **show** *matches* ?γ *m a* (?*p p*) **by** *simp*
qed
qed
qed
qed

private definition *get-exists-matching-src-ips-executable*
:: *iface* ⇒ '*i*::*len common-primitive match-expr* ⇒ '*i wordinterval* **where**
get-exists-matching-src-ips-executable iface m ≡ *let* (*i-matches*, -) = (*primitive-extractor* (*is-Iface*, *iface-sel*) *m*) *in*
if (∀ *is* ∈ *set i-matches*. (*case is of Pos i* ⇒ *match-iface i* (*iface-sel iface*)
| *Neg i* ⇒ ¬*match-iface i* (*iface-sel iface*)))
then
(*let* (*ip-matches*, -) = (*primitive-extractor* (*is-Src*, *src-sel*) *m*) *in*
if ip-matches = []
then
wordinterval-UNIV
else
l2wi-negation-type-intersect (*NegPos-map ipt-iprange-to-interval ip-matches*))
else
Empty-WordInterval

lemma *get-exists-matching-src-ips-executable*:
wordinterval-to-set (*get-exists-matching-src-ips-executable iface m*) = *get-exists-matching-src-ips*

```
iface m
  apply(simp add: get-exists-matching-src-ips-executable-def get-exists-matching-src-ips-def)
   apply(case-tac primitive-extractor (is-Iiface, iiface-sel) m)
   apply(case-tac primitive-extractor (is-Src, src-sel) m)
   apply(simp)
   apply(simp add: l2wi-negation-type-intersect)
   apply(simp add: NegPos-map-simps)
   apply(safe)
       apply(simp-all add: ipt-iprange-to-interval)
     apply(rename-tac i-matches rest1 a b x xa)
     apply(case-tac xa)
      apply(simp-all add: NegPos-set)
      using ipt-iprange-to-interval apply fast+
    apply(rename-tac i-matches rest1 a b x aa ab ba)
    apply(erule-tac x=Pos aa in ballE)
     apply(simp-all add: NegPos-set)
   using NegPos-set(2) by fastforce

 lemma (get-exists-matching-src-ips-executable (Iface ''eth0'')
    (MatchAnd (MatchNot (Match (Src (IpAddrNetmask (ipv4addr-of-dotdecimal
(192,168,0,0)) 24)))) (Match (IIface (Iface ''eth0'')))) =
       RangeUnion (WordInterval 0 0xC0A7FFFF) (WordInterval 0xC0A80100
0xFFFFFFFF) by eval

 private definition get-all-matching-src-ips-executable
  :: iface ⇒ 'i::len common-primitive match-expr ⇒ 'i wordinterval where
  get-all-matching-src-ips-executable iface m ≡ let (i-matches, rest1) = (primitive-extractor
(is-Iiface, iiface-sel) m) in
              if (∀ is ∈ set i-matches. (case is of Pos i ⇒ match-iface i (iface-sel
iface)
                                                  | Neg i ⇒ ¬match-iface i (iface-sel iface)))
            then
              (let (ip-matches, rest2) = (primitive-extractor (is-Src, src-sel) rest1)
in
              if ¬ has-primitive rest2 ∧ matcheq-matchAny rest2
              then
                if ip-matches = []
                then
                  wordinterval-UNIV
                else
                    l2wi-negation-type-intersect (NegPos-map ipt-iprange-to-interval
ip-matches)
              else
                Empty-WordInterval)
            else
              Empty-WordInterval

 lemma get-all-matching-src-ips-executable:
  wordinterval-to-set (get-all-matching-src-ips-executable iface m) = get-all-matching-src-ips
```

```
iface m
  apply(simp add: get-all-matching-src-ips-executable-def get-all-matching-src-ips-def)
   apply(case-tac primitive-extractor (is-Iiface, iiface-sel) m)
   apply(simp, rename-tac i-matches rest1)
   apply(case-tac primitive-extractor (is-Src, src-sel) rest1)
   apply(simp)
   apply(simp add: l2wi-negation-type-intersect)
   apply(simp add: NegPos-map-simps)
   apply(safe)
       apply(simp-all add: ipt-iprange-to-interval)
     apply(rename-tac i-matches rest1 a b x xa)
     apply(case-tac xa)
      apply(simp-all add: NegPos-set)
      using ipt-iprange-to-interval apply fast+
    apply(rename-tac i-matches rest1 a b x aa ab ba)
    apply(erule-tac x=Pos aa in ballE)
     apply(simp-all add: NegPos-set)
   apply(erule-tac x=Neg aa in ballE)
    apply(simp-all add: NegPos-set)
   done
 lemma (get-all-matching-src-ips-executable (Iface ''eth0'')
    (MatchAnd (MatchNot (Match (Src (IpAddrNetmask (ipv4addr-of-dotdecimal
(192,168,0,0)) 24)))) (Match (IIface (Iface ''eth0'')))) =
       RangeUnion (WordInterval 0 0xC0A7FFFF) (WordInterval 0xC0A80100
0xFFFFFFFF) by eval
```

The following algorithm sound but not complete.

```
 private fun no-spoofing-algorithm
   :: iface ⇒ 'i::len ipassignment ⇒ 'i common-primitive rule list ⇒ 'i word set
⇒ 'i word set ⇒ bool where
   no-spoofing-algorithm iface ipassmt [] allowed denied1 ⟷
     (allowed − denied1) ⊆ ipcidr-union-set (set (the (ipassmt iface))) |
    no-spoofing-algorithm iface ipassmt ((Rule m Accept)#rs) allowed denied1 =
no-spoofing-algorithm iface ipassmt rs
       (allowed ∪ get-exists-matching-src-ips iface m) denied1 |
    no-spoofing-algorithm iface ipassmt ((Rule m Drop)#rs) allowed denied1 =
no-spoofing-algorithm iface ipassmt rs
        allowed (denied1 ∪ (get-all-matching-src-ips iface m − allowed)) |
   no-spoofing-algorithm - - - - - = undefined



 private fun no-spoofing-algorithm-executable
   :: iface ⇒ (iface ⇀ ('i::len word × nat) list) ⇒ 'i common-primitive rule list
        ⇒ 'i wordinterval ⇒ 'i wordinterval ⇒ bool where
   no-spoofing-algorithm-executable iface ipassmt [] allowed denied1 ⟷
     wordinterval-subset (wordinterval-setminus allowed denied1) (l2wi (map ip-
cidr-to-interval (the (ipassmt iface)))) |
    no-spoofing-algorithm-executable iface ipassmt ((Rule m Accept)#rs) allowed
```

denied1 = no-spoofing-algorithm-executable iface ipassmt rs
(wordinterval-union allowed (get-exists-matching-src-ips-executable iface m))
denied1 |
no-spoofing-algorithm-executable iface ipassmt ((Rule m Drop)#rs) allowed denied1 = no-spoofing-algorithm-executable iface ipassmt rs
allowed (wordinterval-union denied1 (wordinterval-setminus (get-all-matching-src-ips-executable iface m) allowed)) |
no-spoofing-algorithm-executable - - - - - = undefined

lemma *no-spoofing-algorithm-executable*: *no-spoofing-algorithm-executable iface ipassmt rs allowed denied ⟷*
no-spoofing-algorithm iface ipassmt rs (wordinterval-to-set allowed) (wordinterval-to-set denied)
proof(*induction iface ipassmt rs allowed denied rule*: *no-spoofing-algorithm-executable.induct*)
case (*1 iface ipassmt allowed denied1*)
have (⋃ *a*∈*set* (*the* (*ipassmt iface*)). *case ipcidr-to-interval a of* (*x*, *xa*) ⇒ {*x*..*xa*}) =
(⋃ *x*∈*set* (*the* (*ipassmt iface*)). *uncurry ipset-from-cidr x*)
by(*simp add*: *ipcidr-to-interval-def uncurry-def ipset-from-cidr-ipcidr-to-interval*)
with *1* **show** *?case* **by**(*simp add*: *ipcidr-union-set-uncurry l2wi*)
next
case *2* **thus** *?case* **by**(*simp add*: *get-exists-matching-src-ips-executable get-all-matching-src-ips-executable*)
next
case *3* **thus** *?case* **by**(*simp add*: *get-exists-matching-src-ips-executable get-all-matching-src-ips-executable*)
qed(*simp-all*)

private definition *nospoof TYPE('pkt-ext) iface ipassmt rs* = (∀ *p* :: ('*i*::*len*,'*pkt-ext*) *tagged-packet-scheme*.
(*approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) (*p*⦇*p-iiface*:=*iface-sel iface*⦈) *rs Undecided = Decision FinalAllow*) ⟶
p-src p ∈ (*ipcidr-union-set* (*set* (*the* (*ipassmt iface*)))))
private definition *setbydecision TYPE('pkt-ext) iface rs dec* = {*ip*. ∃ *p* :: ('*i*::*len*,'*pkt-ext*) *tagged-packet-scheme*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*)
(*p*⦇*p-iiface*:=*iface-sel iface*, *p-src* := *ip*⦈) *rs Undecided = Decision dec*}

private lemma *nospoof-setbydecision*:
fixes *rs* :: '*i*::*len common-primitive rule list*
shows *nospoof TYPE('pkt-ext) iface ipassmt rs ⟷*
setbydecision TYPE('pkt-ext) iface rs FinalAllow ⊆ (*ipcidr-union-set* (*set* (*the* (*ipassmt iface*))))
proof
assume *a*: *nospoof TYPE('pkt-ext) iface ipassmt rs*
have *packet-update-iface-simp*: *p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *x*⦈ = *p*⦇*p-src* := *x*, *p-iiface* := *iface-sel iface*⦈
for *p*::('*i*::*len*, '*p*) *tagged-packet-scheme* **and** *x* **by** *simp*

from *a* **show** *setbydecision TYPE('pkt-ext) iface rs FinalAllow ⊆ ipcidr-union-set (set (the (ipassmt iface)))*
apply(*simp add*: *nospoof-def setbydecision-def*)
apply(*safe*)
apply(*rename-tac x p*)
apply(*erule-tac x=p(|p-iiface := iface-sel iface, p-src := x|) in allE*)
apply(*simp*)
apply(*simp add*: *packet-update-iface-simp*)
done
next
assume *a1*: *setbydecision TYPE('pkt-ext) iface rs FinalAllow ⊆ ipcidr-union-set (set (the (ipassmt iface)))*
show *nospoof TYPE('pkt-ext) iface ipassmt rs*
unfolding *nospoof-def*
proof(*safe*)
fix *p* :: (*'i*::*len*,*'pkt-ext*) *tagged-packet-scheme*
assume *a2*: *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) (*p*(|*p-iiface* := *iface-sel iface*|)) *rs Undecided* = *Decision FinalAllow*
— In *setbydecision-fix-p*the ∃ quantifier is gone and we consider this set for *p*.
let *?setbydecision-fix-p*={*ip*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*)
(*p*(|*p-iiface* := *iface-sel iface*, *p-src* := *ip*|)) *rs Undecided* = *Decision FinalAllow*}
from *a1 a2* **have** *1*: *?setbydecision-fix-p ⊆ ipcidr-union-set* (*set* (*the* (*ipassmt iface*))) **by**(*simp add*: *nospoof-def setbydecision-def*) *blast*
from *a2* **have** *2*: *p-src p ∈ ?setbydecision-fix-p* **by** *simp*
from *1 2* **show** *p-src p ∈ ipcidr-union-set* (*set* (*the* (*ipassmt iface*))) **by** *blast*
qed
qed

private definition *setbydecision-all TYPE('pkt-ext) iface rs dec* = {*ip*. ∀ *p* :: (*'i*::*len*,*'pkt-ext*) *tagged-packet-scheme*.
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) (*p*(|*p-iiface*:=*iface-sel iface*, *p-src* := *ip*|)) *rs Undecided* = *Decision dec*}

private lemma *setbydecision-setbydecision-all-Allow*:
(*setbydecision TYPE('pkt-ext) iface rs FinalAllow* − *setbydecision-all TYPE('pkt-ext) iface rs FinalDeny*) =
setbydecision TYPE('pkt-ext) iface rs FinalAllow
apply(*safe*)
apply(*simp add*: *setbydecision-def setbydecision-all-def*)
done
private lemma *setbydecision-setbydecision-all-Deny*:
(*setbydecision TYPE('pkt-ext) iface rs FinalDeny* − *setbydecision-all TYPE('pkt-ext) iface rs FinalAllow*) =
setbydecision TYPE('pkt-ext) iface rs FinalDeny

apply(*safe*)
apply(*simp add*: *setbydecision-def setbydecision-all-def*)
done

private lemma *setbydecision-append*:
simple-ruleset (*rs1* @ *rs2*) ⟹
setbydecision TYPE('*pkt-ext*) *iface* (*rs1* @ *rs2*) *FinalAllow* =
setbydecision TYPE('*pkt-ext*) *iface rs1 FinalAllow* ∪ {*ip*. ∃ *p* :: ('*i*::*len*,'*pkt-ext*) *tagged-packet-scheme*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*)

(*p*⦇*p-iiface*:=*iface-sel iface*, *p-src* := *ip*⦈) *rs2 Undecided* = *Decision FinalAllow* ∧
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) (*p*⦇*p-iiface*:=*iface-sel iface*, *p-src* := *ip*⦈) *rs1 Undecided* = *Undecided*}
apply(*simp add*: *setbydecision-def*)
apply(*subst Set.Collect-disj-eq*[*symmetric*])
apply(*rule Set.Collect-cong*)
apply(*subst approximating-bigstep-fun-seq-Undecided-t-wf*)
apply(*simp add*: *simple-imp-good-ruleset good-imp-wf-ruleset*)
by *blast*

private lemma *not-FinalAllow*: *foo* ≠ *Decision FinalAllow* ⟷ *foo* = *Decision FinalDeny* ∨ *foo* = *Undecided*
apply(*cases foo*)
apply *simp-all*
apply(*rename-tac x2*)
apply(*case-tac x2*)
apply(*simp-all*)
done

private lemma *setbydecision-all-appendAccept*: *simple-ruleset* (*rs* @ [*Rule r Accept*]) ⟹
setbydecision-all TYPE('*pkt-ext*) *iface rs FinalDeny* = *setbydecision-all TYPE*('*pkt-ext*) *iface* (*rs* @ [*Rule r Accept*]) *FinalDeny*
apply(*simp add*: *setbydecision-all-def*)
apply(*rule Set.Collect-cong*)
apply(*subst approximating-bigstep-fun-seq-Undecided-t-wf*)
apply(*simp add*: *simple-imp-good-ruleset good-imp-wf-ruleset*)
apply(*simp add*: *not-FinalAllow*)
done

private lemma *setbydecision-all-append-subset*: *simple-ruleset* (*rs1* @ *rs2*) ⟹
setbydecision-all TYPE('*pkt-ext*) *iface rs1 FinalDeny* ∪ {*ip*. ∀ *p* :: ('*i*::*len*,'*pkt-ext*) *tagged-packet-scheme*.
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) (*p*⦇*p-iiface*:=*iface-sel iface*, *p-src* := *ip*⦈) *rs2 Undecided* = *Decision FinalDeny* ∧
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) (*p*⦇*p-iiface*:=*iface-sel iface*, *p-src* := *ip*⦈) *rs1 Undecided* = *Undecided*}
⊆

setbydecision-all TYPE('pkt-ext) iface (rs1 @ rs2) FinalDeny
unfolding *setbydecision-all-def*
apply(*subst Set.Collect-disj-eq*[*symmetric*])
apply(*rule Set.Collect-mono*)
apply(*subst approximating-bigstep-fun-seq-Undecided-t-wf*)
apply(*simp add*: *simple-imp-good-ruleset good-imp-wf-ruleset*)
apply(*simp add*: *not-FinalAllow*)
done

private lemma *setbydecision-all TYPE('pkt-ext) iface rs1 FinalDeny* ∪
{*ip*. ∀ *p* :: ('*i*::*len*,'*pkt-ext*) *tagged-packet-scheme*.
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) (*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *ip*⦈) *rs1 Undecided* = *Undecided*}
⊆
− *setbydecision TYPE('pkt-ext) iface rs1 FinalAllow*
unfolding *setbydecision-all-def*
unfolding *setbydecision-def*
apply(*subst Set.Collect-neg-eq*[*symmetric*])
apply(*subst Set.Collect-disj-eq*[*symmetric*])
apply(*rule Set.Collect-mono*)
by(*simp*)

private lemma *Collect-minus-eq*: {*x*. *P x*} − {*x*. *Q x*} = {*x*. *P x* ∧ ¬ *Q x*} **by** *blast*
private lemma *setbydecision-all-append-subset2*:
simple-ruleset (*rs1* @ *rs2*) ⟹
setbydecision-all TYPE('pkt-ext) iface rs1 FinalDeny ∪
(*setbydecision-all TYPE('pkt-ext) iface rs2 FinalDeny* −
setbydecision TYPE('pkt-ext) iface rs1 FinalAllow)
⊆ *setbydecision-all TYPE('pkt-ext) iface* (*rs1* @ *rs2*) *FinalDeny*
unfolding *setbydecision-all-def*
unfolding *setbydecision-def*
apply(*subst Collect-minus-eq*)
apply(*subst Set.Collect-disj-eq*[*symmetric*])
apply(*rule Set.Collect-mono*)
apply(*subst approximating-bigstep-fun-seq-Undecided-t-wf*)
apply(*simp add*: *simple-imp-good-ruleset good-imp-wf-ruleset*; *fail*)
apply(*intro impI allI*)
apply(*simp add*: *not-FinalAllow*)
apply(*case-tac approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) (*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *x*⦈) *rs1 Undecided*)
subgoal by(*elim disjE*) *simp-all*
apply(*rename-tac x2*)
apply(*case-tac x2*)
prefer *2*
apply *simp*
apply(*elim disjE*)
apply(*simp*)

by *blast*

private lemma *setbydecision-all TYPE('pkt-ext) iface rs FinalDeny ⊆ − setbydecision TYPE('pkt-ext) iface rs FinalAllow*
apply(*simp add*: *setbydecision-def setbydecision-all-def*)
apply(*subst Set.Collect-neg-eq*[*symmetric*])
apply(*rule Set.Collect-mono*)
apply(*simp*)
done

private lemma *no-spoofing-algorithm-sound-generalized*:
fixes *rs1* :: *'i*::*len common-primitive rule list*
shows *simple-ruleset rs1* ⟹ *simple-ruleset rs2* ⟹
(∀ *r* ∈ *set rs2*. *normalized-nnf-match* (*get-match r*)) ⟹
setbydecision TYPE('pkt-ext) iface rs1 FinalAllow ⊆ *allowed* ⟹
denied1 ⊆ *setbydecision-all TYPE('pkt-ext) iface rs1 FinalDeny* ⟹
no-spoofing-algorithm iface ipassmt rs2 allowed denied1 ⟹
nospoof TYPE('pkt-ext) iface ipassmt (*rs1*@*rs2*)
proof(*induction iface ipassmt rs2 allowed denied1 arbitrary*: *rs1 allowed denied1 rule*: *no-spoofing-algorithm.induct*)
case (*1 iface ipassmt*)
from *1* **have** *allowed − denied1* ⊆ *ipcidr-union-set* (*set* (*the* (*ipassmt iface*)))
by(*simp*)
with *1* **have** *setbydecision TYPE('pkt-ext) iface rs1 FinalAllow − setbydecision-all TYPE('pkt-ext) iface rs1 FinalDeny*
⊆ *ipcidr-union-set* (*set* (*the* (*ipassmt iface*)))
by *blast*
thus *?case*
by(*simp add*: *nospoof-setbydecision setbydecision-setbydecision-all-Allow*)
next
case (*2 iface ipassmt m rs*)
from *2*(*2*) **have** *simple-rs1*: *simple-ruleset rs1* **by**(*simp add*: *simple-ruleset-def*)
hence *simple-rs'*: *simple-ruleset* (*rs1* @ [*Rule m Accept*]) **by**(*simp add*: *simple-ruleset-def*)
from *2*(*3*) **have** *simple-rs*: *simple-ruleset rs* **by**(*simp add*: *simple-ruleset-def*)
with *2* **have** *IH*: ⋀*rs' allowed denied1*.
simple-ruleset rs' ⟹
setbydecision TYPE('pkt-ext) iface rs' FinalAllow ⊆ *allowed* ⟹
denied1 ⊆ *setbydecision-all TYPE('pkt-ext) iface rs' FinalDeny* ⟹
no-spoofing-algorithm iface ipassmt rs allowed denied1 ⟹ *nospoof TYPE('pkt-ext) iface ipassmt* (*rs'* @ *rs*)
by(*simp*)
from *2*(*5*) **have** *setbydecision TYPE('pkt-ext) iface* (*rs1* @ [*Rule m Accept*]) *FinalAllow* ⊆
(*allowed* ∪ {*ip*. ∃ *p* :: (*'i*::*len*,*'pkt-ext*) *tagged-packet-scheme*. *matches* (*common-matcher*, *in-doubt-allow*) *m Accept* (*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *ip*⦈)})
apply(*simp add*: *setbydecision-append*[*OF simple-rs'*])
by *blast*
with *get-exists-matching-src-ips-subset 2*(*4*) **have** *allowed*: *setbydecision TYPE('pkt-ext)*

iface (*rs1* @ [*Rule m Accept*]) *FinalAllow* ⊆ (*allowed* ∪ *get-exists-matching-src-ips iface m*)
by *fastforce*

from *2*(*6*) *setbydecision-all-appendAccept*[*OF simple-rs'*, **where** *'pkt-ext* = *'pkt-ext*] **have** *denied1*:
denied1 ⊆ *setbydecision-all TYPE*('*pkt-ext*) *iface* (*rs1* @ [*Rule m Accept*]) *FinalDeny* **by** *simp*

from *2*(*7*) **have** *no-spoofing-algorithm-prems*: *no-spoofing-algorithm iface ipassmt rs*
(*allowed* ∪ *get-exists-matching-src-ips iface m*) *denied1*
by(*simp*)

from *IH*[*OF simple-rs' allowed denied1 no-spoofing-algorithm-prems*] **have** *nospoof TYPE*('*pkt-ext*) *iface ipassmt* ((*rs1* @ [*Rule m Accept*]) @ *rs*) **by** *blast*
thus *?case* **by**(*simp*)
next
case (*3 iface ipassmt m rs*)
from *3*(*2*) **have** *simple-rs1*: *simple-ruleset rs1* **by**(*simp add*: *simple-ruleset-def*)
hence *simple-rs'*: *simple-ruleset* (*rs1* @ [*Rule m Drop*]) **by**(*simp add*: *simple-ruleset-def*)
from *3*(*3*) **have** *simple-rs*: *simple-ruleset rs* **by**(*simp add*: *simple-ruleset-def*)
with *3* **have** *IH*: ⋀*rs' allowed denied1*.
simple-ruleset rs' ⟹
setbydecision TYPE('*pkt-ext*) *iface rs' FinalAllow* ⊆ *allowed* ⟹
denied1 ⊆ *setbydecision-all TYPE*('*pkt-ext*) *iface rs' FinalDeny* ⟹
no-spoofing-algorithm iface ipassmt rs allowed denied1 ⟹ *nospoof TYPE*('*pkt-ext*) *iface ipassmt* (*rs'* @ *rs*)
by(*simp*)
from *3*(*5*) *simple-rs'* **have** *allowed*: *setbydecision TYPE*('*pkt-ext*) *iface* (*rs1* @ [*Rule m Drop*]) *FinalAllow* ⊆ *allowed*
by(*simp add*: *setbydecision-append*)

have {*ip*. ∀ *p* :: ('*i*,'*pkt-ext*) *tagged-packet-scheme*. *matches* (*common-matcher*, *in-doubt-allow*) *m Drop* (*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *ip*⦈)} ⊆
setbydecision-all TYPE('*pkt-ext*) *iface* [*Rule m Drop*] *FinalDeny* **by**(*simp add*: *setbydecision-all-def*)
with *3*(*5*) **have** *setbydecision-all TYPE*('*pkt-ext*) *iface rs1 FinalDeny* ∪ ({*ip*. ∀ *p* :: ('*i*,'*pkt-ext*) *tagged-packet-scheme*. *matches* (*common-matcher*, *in-doubt-allow*) *m Drop* (*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *ip*⦈)} − *allowed*) ⊆
setbydecision-all TYPE('*pkt-ext*) *iface rs1 FinalDeny* ∪ (*setbydecision-all TYPE*('*pkt-ext*) *iface* [*Rule m Drop*] *FinalDeny* − *setbydecision TYPE*('*pkt-ext*) *iface rs1 FinalAllow*)
by *blast*
with *3*(*6*) *setbydecision-all-append-subset2*[*OF simple-rs'*, *of iface*] **have**
denied1 ∪ ({*ip*. ∀ *p* :: ('*i*,'*pkt-ext*) *tagged-packet-scheme*. *matches* (*common-matcher*, *in-doubt-allow*) *m Drop* (*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *ip*⦈)} − *allowed*) ⊆
setbydecision-all TYPE('*pkt-ext*) *iface* (*rs1* @ [*Rule m Drop*]) *FinalDeny*

by *blast*
with *get-all-matching-src-ips 3(4)* **have** *denied1*:
denied1 ∪ (*get-all-matching-src-ips iface m* − *allowed*) ⊆ *setbydecision-all TYPE('pkt-ext) iface* (*rs1* @ [*Rule m Drop*]) *FinalDeny*
by *force*

from *3(7)* **have** *no-spoofing-algorithm-prems*: *no-spoofing-algorithm iface ipassmt rs allowed*
(*denied1* ∪ (*get-all-matching-src-ips iface m* − *allowed*))
apply(*simp*)
done

from *IH*[*OF simple-rs' allowed denied1 no-spoofing-algorithm-prems*] **have** *nospoof TYPE('pkt-ext) iface ipassmt* ((*rs1* @ [*Rule m Drop*]) @ *rs*) **by** *blast*
thus *?case* **by**(*simp*)
next
case *4-1* **thus** *?case* **by**(*simp add*: *simple-ruleset-def*)
next
case *4-2* **thus** *?case* **by**(*simp add*: *simple-ruleset-def*)
next
case *4-3* **thus** *?case* **by**(*simp add*: *simple-ruleset-def*)
next
case *4-4* **thus** *?case* **by**(*simp add*: *simple-ruleset-def*)
next
case *4-5* **thus** *?case* **by**(*simp add*: *simple-ruleset-def*)
next
case *4-6* **thus** *?case* **by**(*simp add*: *simple-ruleset-def*)
next
case *4-7* **thus** *?case* **by**(*simp add*: *simple-ruleset-def*)
qed

definition *no-spoofing-iface* :: *iface* ⇒ *'i*::*len ipassignment* ⇒ *'i common-primitive rule list* ⇒ *bool* **where**
no-spoofing-iface iface ipassmt rs ≡ *no-spoofing-algorithm iface ipassmt rs* {} {}

lemma[*code*]: *no-spoofing-iface iface ipassmt rs* =
no-spoofing-algorithm-executable iface ipassmt rs Empty-WordInterval Empty-WordInterval
by(*simp add*: *no-spoofing-iface-def no-spoofing-algorithm-executable*)

private corollary *no-spoofing-algorithm-sound*: *simple-ruleset rs* ⟹ ∀ *r*∈*set rs*. *normalized-nnf-match* (*get-match r*) ⟹
no-spoofing-iface iface ipassmt rs ⟹ *nospoof TYPE('pkt-ext) iface ipassmt rs*
unfolding *no-spoofing-iface-def*
apply(*rule no-spoofing-algorithm-sound-generalized*[*of* [] *rs iface* {} {}, *simplified*])
apply(*simp-all*)
apply(*simp add*: *simple-ruleset-def*)
apply(*simp add*: *setbydecision-def*)

done

The *nospoof* definition used throughout the proofs corresponds to checking *no-spoofing* for all interfaces

private lemma *nospoof*: *simple-ruleset rs* $\Longrightarrow$ ($\forall$ *iface* $\in$ *dom ipassmt*. *nospoof TYPE*('*pkt-ext*) *iface ipassmt rs*) $\longleftrightarrow$ *no-spoofing TYPE*('*pkt-ext*) *ipassmt rs*
unfolding *nospoof-def no-spoofing-def*
apply(*drule simple-imp-good-ruleset*)
apply(*subst approximating-semantics-iff-fun-good-ruleset*)
apply(*simp-all*)
done

theorem *no-spoofing-iface*: *simple-ruleset rs* $\Longrightarrow$ $\forall$ *r*$\in$*set rs*. *normalized-nnf-match* (*get-match r*) $\Longrightarrow$
$\forall$ *iface* $\in$ *dom ipassmt*. *no-spoofing-iface iface ipassmt rs* $\Longrightarrow$ *no-spoofing TYPE*('*pkt-ext*) *ipassmt rs*
by(*auto dest*: *nospoof no-spoofing-algorithm-sound*)

Examples

Example 1: Ruleset: Accept all non-spoofed packets, drop rest.

lemma *no-spoofing-iface*
(*Iface* ''*eth0*'')
[*Iface* ''*eth0*'' $\mapsto$ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*)) *24*))) (*Match* (*IIface* (*Iface* ''*eth0*''))) *action.Accept*,
Rule MatchAny action.Drop] **by** *eval*
lemma *no-spoofing TYPE*('*pkt-ext*)
[*Iface* ''*eth0*'' $\mapsto$ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*)) *24*))) (*Match* (*IIface* (*Iface* ''*eth0*''))) *action.Accept*,
Rule MatchAny action.Drop]
apply(*rule no-spoofing-iface*)
apply(*simp-all add*: *simple-ruleset-def*)
by *eval*

Example 2: Ruleset: Drop packets from a spoofed IP range, allow rest. Handles negated interfaces correctly.

lemma *no-spoofing TYPE*('*pkt-ext*)
[*Iface* ''*eth0*'' $\mapsto$ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*Match* (*IIface* (*Iface* ''*wlan+*''))) (*Match* (*Extra* ''*no idea what this is*''))) *action.Accept*, — not interesting for spoofing
Rule (*MatchNot* (*Match* (*IIface* (*Iface* ''*eth0+*'')))) *action.Accept*, — not interesting for spoofing
Rule (*MatchAnd* (*MatchNot* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*)) *24*)))) (*Match* (*IIface* (*Iface* ''*eth0*'')))) *action.Drop*, — spoof-protect here
Rule MatchAny action.Accept]

apply(*rule no-spoofing-iface*)
apply(*simp-all add*: *simple-ruleset-def*)
by *eval*

Example 3: Accidentally, matching on wlan+, spoofed packets for eth0 are allowed. First, we prove that there actually is no spoofing protection. Then we show that our algorithm finds out.

lemma ¬ *no-spoofing TYPE*('*pkt-ext*)
[*Iface* ''*eth0*'' ↦ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchNot* (*Match* (*IIface* (*Iface* ''*wlan+*'')))) *action.Accept*, —
accidently allow everything for eth0
Rule (*MatchAnd* (*MatchNot* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal*
(*192*,*168*,*0*,*0*)) *24*)))) (*Match* (*IIface* (*Iface* ''*eth0*'')))) *action.Drop*,
Rule MatchAny action.Accept]

apply(*simp add*: *no-spoofing-def*)
apply(*rule-tac x*=*p*(|*p-src* := *0*|) **in** *exI*)
apply(*simp add*: *range-0-max-UNIV ipcidr-union-set-def*)
apply(*intro conjI*)
apply(*subst approximating-semantics-iff-fun-good-ruleset*)
apply(*simp add*: *good-ruleset-def*; *fail*)
apply(*simp add*: *bunch-of-lemmata-about-matches*
match-simplematcher-SrcDst-not
primitive-matcher-generic.Iface-single[*OF primitive-matcher-generic-common-matcher*]
primitive-matcher-generic.Iface-single-not[*OF primitive-matcher-generic-common-matcher*])
apply *eval*
done

lemma ¬ *no-spoofing-iface*
(*Iface* ''*eth0*'')
[*Iface* ''*eth0*'' ↦ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchNot* (*Match* (*IIface* (*Iface* ''*wlan+*'')))) *action.Accept*, —
accidently allow everything for eth0
Rule (*MatchAnd* (*MatchNot* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal*
(*192*,*168*,*0*,*0*)) *24*)))) (*Match* (*IIface* (*Iface* ''*eth0*'')))) *action.Drop*,
Rule MatchAny action.Accept]
by *eval*

Example 4: Ruleset: Drop packets coming from the wrong interface, allow the rest. Warning: this does not prevent spoofing for eth0! Explanation: someone on eth0 can send a packet e.g. with source IP 8.8.8.8 The ruleset only prevents spoofing of 192.168.0.0/24 for other interfaces

lemma ¬ *no-spoofing TYPE*('*pkt-ext*) [*Iface* ''*eth0*'' ↦ [(*ipv4addr-of-dotdecimal*
(*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal*
(*192*,*168*,*0*,*0*)) *24*))) (*MatchNot* (*Match* (*IIface* (*Iface* ''*eth0*'')))))) *action.Drop*,
Rule MatchAny action.Accept]

apply(*simp add*: *no-spoofing-def*)
apply(*rule-tac x*=*p*(|*p-src* := *0*|) **in** *exI*)
apply(*simp add*: *range-0-max-UNIV ipcidr-union-set-def*)
apply(*intro conjI*)
apply(*subst approximating-semantics-iff-fun-good-ruleset*)
apply(*simp add*: *good-ruleset-def*; *fail*)
apply(*simp add*: *bunch-of-lemmata-about-matches*
primitive-matcher-generic.Iface-single[*OF primitive-matcher-generic-common-matcher*]
primitive-matcher-generic.Iface-single-not[*OF primitive-matcher-generic-common-matcher*])
apply *eval*
done

Our algorithm detects it.

lemma ¬ *no-spoofing-iface*
(*Iface ''eth0''*)
[*Iface ''eth0''* ↦ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal*
(*192*,*168*,*0*,*0*)) *24*))) (*MatchNot* (*Match* (*IIface* (*Iface ''eth0''*))))) *action.Drop*,
Rule MatchAny action.Accept] **by** *eval*

Example 5: Spoofing protection but the algorithm fails. The algorithm *no-spoofing-iface* is only sound, not complete. The ruleset first drops spoofed packets for TCP and then drops spoofed packets for ¬ *TCP*. The algorithm cannot detect that $TCP \cup \neg TCP$ together will match all spoofed packets.

lemma *no-spoofing TYPE*('*pkt-ext*) [*Iface ''eth0''* ↦ [(*ipv4addr-of-dotdecimal*
(*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*MatchNot* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal*
(*192*,*168*,*0*,*0*)) *24*))))
(*MatchAnd* (*Match* (*IIface* (*Iface ''eth0''*)))
(*Match* (*Prot* (*Proto TCP*))))) *action.Drop*,
Rule (*MatchAnd* (*MatchNot* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal*
(*192*,*168*,*0*,*0*)) *24*))))
(*MatchAnd* (*Match* (*IIface* (*Iface ''eth0''*)))
(*MatchNot* (*Match* (*Prot* (*Proto TCP*)))))) *action.Drop*,
Rule MatchAny action.Accept] (**is** *no-spoofing TYPE*('*pkt-ext*) *?ipassmt*
?rs)
proof −
have *1*: ∀ *p*. (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*?rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ⟷
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) *p ?rs*
Undecided = *Decision FinalAllow*
by(*subst approximating-semantics-iff-fun-good-ruleset*) (*simp-all add*: *good-ruleset-def*)
show *?thesis*
unfolding *no-spoofing-def*
apply(*simp add*: *1 ipcidr-union-set-def*)
apply(*simp add*: *bunch-of-lemmata-about-matches*
primitive-matcher-generic.Iface-single[*OF primitive-matcher-generic-common-matcher*]
primitive-matcher-generic.Iface-single-not[*OF primitive-matcher-generic-common-matcher*])
apply(*simp add*: *match-iface.simps match-simplematcher-SrcDst-not*

primitive-matcher-generic.Prot-single[*OF primitive-matcher-generic-common-matcher*]
primitive-matcher-generic.Prot-single-not[*OF primitive-matcher-generic-common-matcher*])
done
qed

Spoofing protection but the algorithm cannot certify spoofing protection.

lemma ¬ *no-spoofing-iface*
(*Iface ''eth0''*)
[*Iface ''eth0''* ↦ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*MatchNot* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*)) *24*))))
(*MatchAnd* (*Match* (*IIface* (*Iface ''eth0''*)))
(*Match* (*Prot* (*Proto TCP*))))) *action.Drop*,
Rule (*MatchAnd* (*MatchNot* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*)) *24*))))
(*MatchAnd* (*Match* (*IIface* (*Iface ''eth0''*)))
(*MatchNot* (*Match* (*Prot* (*Proto TCP*)))))) *action.Drop*,
Rule MatchAny action.Accept] **by** *eval*

end

lemma *no-spoofing-iface* (*Iface ''eth1.1011''*)
([*Iface ''eth1.1011''* ↦ [(*ipv4addr-of-dotdecimal* (*131*,*159*,*14*,*0*), *24*)]]:: *32 ipassignment*)
[*Rule* (*MatchNot* (*Match* (*IIface* (*Iface ''eth1.1011+''*)))) *action.Accept*,
Rule (*MatchAnd* (*MatchNot* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*131*,*159*,*14*,*0*)) *24*)))) (*Match* (*IIface* (*Iface ''eth1.1011''*)))) *action.Drop*,
Rule MatchAny action.Accept] **by** *eval*

We only check accepted packets. If there is no default rule (this will never happen if parsed from iptables!), the result is unfinished.

lemma *no-spoofing-iface* (*Iface ''eth1.1011''*)
([*Iface ''eth1.1011''* ↦ [(*ipv4addr-of-dotdecimal* (*131*,*159*,*14*,*0*), *24*)]]:: *32 ipassignment*)
[*Rule* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*127*, *0*, *0*, *0*)) *8*))) *Drop*] **by** *eval*

end
theory *Common-Primitive-toString*
imports *Simple-Firewall.Primitives-toString*
Common-Primitive-Matcher
begin

33 Firewall toString Functions

fun *ipt-ipv4range-toString* :: *32 ipt-iprange* ⇒ *string* **where**
ipt-ipv4range-toString (*IpAddr ip*) = *ipv4addr-toString ip* |
ipt-ipv4range-toString (*IpAddrNetmask ip n*) = *ipv4addr-toString ip*@*''/''*@*string-of-nat n* |

ipt-ipv4range-toString (IpAddrRange ip1 ip2) = ipv4addr-toString ip1@''−''@ipv4addr-toString ip2

fun *ipt-ipv6range-toString* :: *128 ipt-iprange* ⇒ *string* **where**
ipt-ipv6range-toString (IpAddr ip) = ipv6addr-toString ip |
ipt-ipv6range-toString (IpAddrNetmask ip n) = ipv6addr-toString ip@''/''@string-of-nat n |
ipt-ipv6range-toString (IpAddrRange ip1 ip2) = ipv6addr-toString ip1@''−''@ipv6addr-toString ip2

definition *ipv4addr-wordinterval-pretty-toString* :: *32 wordinterval* ⇒ *string* **where**
ipv4addr-wordinterval-pretty-toString wi = list-toString ipt-ipv4range-toString (wi-to-ipt-iprange wi)

lemma *ipv4addr-wordinterval-pretty-toString*
(*RangeUnion* (*RangeUnion* (*WordInterval 0x7F000000 0x7FFFFFFF*) (*WordInterval 0x1020304 0x1020306*))
(*WordInterval 0x8080808 0x8080808*)) = ''[*127.0.0.0/8*, *1.2.3.4−1.2.3.6*, *8.8.8.8*]'' **by** *eval*

fun *action-toString* :: *action* ⇒ *string* **where**
action-toString action.Accept = ''−*j ACCEPT*'' |
action-toString action.Drop = ''−*j DROP*'' |
action-toString action.Reject = ''−*j REJECT*'' |
action-toString (*action.Call target*) = ''−*j* ''@*target*@'' (*call*)'' |
action-toString (*action.Goto target*) = ''−*g* ''@*target* |
action-toString action.Empty = '''' |
action-toString action.Log = ''−*j LOG*'' |
action-toString action.Return = ''−*j RETURN*'' |
action-toString action.Unknown = ''!!!!!!!!!!! *UNKNOWN* !!!!!!!!!!!''

fun *common-primitive-toString* :: ('*i*::*len word* ⇒ *string*) ⇒ '*i common-primitive* ⇒ *string* **where**
common-primitive-toString ipToStr (*Src* (*IpAddr ip*)) = ''−*s* ''@*ipToStr ip* |
common-primitive-toString ipToStr (*Dst* (*IpAddr ip*)) = ''−*d* ''@*ipToStr ip* |
common-primitive-toString ipToStr (*Src* (*IpAddrNetmask ip n*)) = ''−*s* ''@*ipToStr ip*@''/''@*string-of-nat n* |
common-primitive-toString ipToStr (*Dst* (*IpAddrNetmask ip n*)) = ''−*d* ''@*ipToStr ip*@''/''@*string-of-nat n* |
common-primitive-toString ipToStr (*Src* (*IpAddrRange ip1 ip2*)) = ''−*m iprange* −−*src−range* ''@*ipToStr ip1*@''−''@*ipToStr ip2* |
common-primitive-toString ipToStr (*Dst* (*IpAddrRange ip1 ip2*)) = ''−*m iprange* −−*dst−range* ''@*ipToStr ip1*@''−''@*ipToStr ip2* |
common-primitive-toString - (*IIface ifce*) = *iface-toString* ''−*i* '' *ifce* |
common-primitive-toString - (*OIface ifce*) = *iface-toString* ''−*o* '' *ifce* |
common-primitive-toString - (*Prot prot*) = ''−*p* ''@*protocol-toString prot* |

common-primitive-toString - (*Src-Ports* (*L4Ports prot pts*)) = ''−m ''@*primitive-protocol-toString prot*@'' −−*spts* '' @ *list-toString* (*ports-toString* '''') *pts* |
common-primitive-toString - (*Dst-Ports* (*L4Ports prot pts*)) = ''−m ''@*primitive-protocol-toString prot*@'' −−*dpts* '' @ *list-toString* (*ports-toString* '''') *pts* |
common-primitive-toString - (*MultiportPorts* (*L4Ports prot pts*)) = ''−p ''@*primitive-protocol-toString prot*@'' −m *multiport* −−*ports* '' @ *list-toString* (*ports-toString* '''') *pts* |
common-primitive-toString - (*CT-State S*) = ''−m *state* −−*state* ''@*ctstate-set-toString S* |
common-primitive-toString - (*L4-Flags* (*TCP-Flags c m*)) = ''−−*tcp*−*flags* ''@*ipt-tcp-flags-toString c*@'' ''@*ipt-tcp-flags-toString m* |
common-primitive-toString - (*Extra e*) = ''~~''@*e*@''~~''

definition *common-primitive-ipv4-toString* :: *32 common-primitive* ⇒ *string* **where**
common-primitive-ipv4-toString ≡ *common-primitive-toString ipv4addr-toString*

definition *common-primitive-ipv6-toString* :: *128 common-primitive* ⇒ *string* **where**
common-primitive-ipv6-toString ≡ *common-primitive-toString ipv6addr-toString*

fun *common-primitive-match-expr-toString*
:: (*'i common-primitive* ⇒ *string*) ⇒ *'i common-primitive match-expr* ⇒ *string*
where
common-primitive-match-expr-toString toStr MatchAny = '''' |
common-primitive-match-expr-toString toStr (*Match m*) = *toStr m* |
common-primitive-match-expr-toString toStr (*MatchAnd m1 m2*) =
common-primitive-match-expr-toString toStr m1 @'' '' @ *common-primitive-match-expr-toString toStr m2* |
common-primitive-match-expr-toString toStr (*MatchNot* (*Match m*)) = ''! ''@*toStr m* |
common-primitive-match-expr-toString toStr (*MatchNot m*) = ''*NOT* (''@*common-primitive-match-expr-toStr toStr m*@'')''

definition *common-primitive-match-expr-ipv4-toString* :: *32 common-primitive match-expr* ⇒ *string* **where**
common-primitive-match-expr-ipv4-toString ≡ *common-primitive-match-expr-toString common-primitive-ipv4-toString*

definition *common-primitive-match-expr-ipv6-toString* :: *128 common-primitive match-expr* ⇒ *string* **where**
common-primitive-match-expr-ipv6-toString ≡ *common-primitive-match-expr-toString common-primitive-ipv6-toString*

fun *common-primitive-rule-toString* :: *32 common-primitive rule* ⇒ *string* **where**
common-primitive-rule-toString (*Rule m a*) = *common-primitive-match-expr-ipv4-toString m* @'' ''@*action-toString a*

end

34 Routing and IP Assignments

theory *Routing-IpAssmt*
imports *Ipassmt*
Routing.Routing-Table
begin
context
begin

34.1 Routing IP Assignment

Up to now, the definitions were all still on word intervals because those are much more convenient to work with.

definition *routing-ipassmt* :: *'i*::*len routing-rule list* ⇒ (*iface* × (*'i word* × *nat*) *list*) *list*
where
routing-ipassmt rt ≡ *map* (*apfst Iface* ∘ *apsnd cidr-split*) (*routing-ipassmt-wi rt*)

private lemma *ipcidr-union-cidr-split*[*simp*]: *ipcidr-union-set* (*set* (*cidr-split x*)) = *wordinterval-to-set x*
apply(*subst cidr-split-prefix*[*symmetric*])
apply(*fact ipcidr-union-set-uncurry*)
done

private lemma *map-of-map-Iface*: *map-of* (*map* (λ*x*. (*Iface* (*fst x*), *f* (*snd x*))) *xs*) (*Iface ifce*) =
map-option f ((*map-of xs*) *ifce*)
by (*induct xs*) (*auto*)

lemma *routing-ipassmt-wi* ([]::*32 prefix-routing*) = [(*output-iface* (*routing-action* (*undefined* :: *32 routing-rule*)), *WordInterval 0 0xFFFFFFFF*)]
by *code-simp*

lemma *routing-ipassmt*:
valid-prefixes rt ⟹
output-iface (*routing-table-semantics rt* (*p-dst p*)) = *p-oiface p* ⟹
∃ *p-ips*. *map-of* (*routing-ipassmt rt*) (*Iface* (*p-oiface p*)) = *Some p-ips* ∧ *p-dst p* ∈ *ipcidr-union-set* (*set p-ips*)
apply(*simp add*: *routing-ipassmt-def*)
apply(*drule routing-ipassmt-wi*[**where** *output-port*=*p-oiface p* **and** *k*=*p-dst p*])
apply(*simp*)
apply(*elim exE*, *rename-tac ip-range*)
apply(*rule-tac x*=*cidr-split ip-range* **in** *exI*)
apply(*simp*)
apply(*simp add*: *comp-def*)
apply(*simp add*: *map-of-map-Iface*)
apply(*rule-tac x*=*ip-range* **in** *exI*)
apply(*simp*)

by (*simp add*: *routing-ipassmt-wi-distinct*)

lemma *routing-ipassmt-ipassmt-sanity-disjoint*: *valid-prefixes* (*rt*::('*i*::*len*) *prefix-routing*)
⟹
ipassmt-sanity-disjoint (*map-of* (*routing-ipassmt rt*))
unfolding *ipassmt-sanity-disjoint-def routing-ipassmt-def comp-def*
apply(*clarsimp*)
apply(*drule map-of-SomeD*)+
apply(*clarsimp split*: *iface.splits*)
using *routing-ipassmt-wi-disjoint*[**where** *'i* = *'i*] **by** *meson*

lemma *routing-ipassmt-distinct*: *distinct* (*map fst* (*routing-ipassmt rtbl*))
using *routing-ipassmt-wi-distinct*[*of rtbl*]
unfolding *routing-ipassmt-def*
apply(*simp add*: *comp-def*)
apply(*subst distinct-map*[**where** *f* = *Iface* **and** *xs* = *map fst* (*routing-ipassmt-wi rtbl*), *simplified*, *unfolded comp-def*])
apply(*auto intro*: *inj-onI*)
done

end

end
theory *Output-Interface-Replace*
imports
Ipassmt
Routing-IpAssmt
Common-Primitive-toString
begin

35 Replacing output interfaces by their IP ranges according to Routing

Copy of `Interface_Replace.thy`

definition *ipassmt-iface-replace-dstip-mexpr*
:: *'i*::*len ipassignment* ⇒ *iface* ⇒ *'i common-primitive match-expr* **where**
ipassmt-iface-replace-dstip-mexpr ipassmt ifce ≡ *case ipassmt ifce of*
None ⇒ *Match* (*OIface ifce*)
| *Some ips* ⇒ (*match-list-to-match-expr* (*map* (*Match* ∘ *Dst*) (*map* (*uncurry IpAddrNetmask*) *ips*)))

lemma *matches-ipassmt-iface-replace-dstip-mexpr*:
matches (*common-matcher*, α) (*ipassmt-iface-replace-dstip-mexpr ipassmt ifce*) *a p* ⟷ (*case ipassmt ifce of*
None ⇒ *match-iface ifce* (*p-oiface p*)
| *Some ips* ⇒ *p-dst p* ∈ *ipcidr-union-set* (*set ips*)
)
proof(*cases ipassmt ifce*)

case *None* **thus** *?thesis* **by**(*simp add*: *ipassmt-iface-replace-dstip-mexpr-def primitive-matcher-generic.Iface-single*[*OF primitive-matcher-generic-common-matcher*])
next
case (*Some ips*)
have *matches* (*common-matcher*, *α*) (*match-list-to-match-expr* (*map* (*Match* ∘ *Dst* ∘ (*uncurry IpAddrNetmask*)) *ips*)) *a p* ⟷
(∃ *m*∈*set ips*. *p-dst p* ∈ (*uncurry ipset-from-cidr m*))
by(*simp add*: *match-list-to-match-expr-disjunction*[*symmetric*]
match-list-matches match-simplematcher-SrcDst ipt-iprange-to-set-uncurry-IpAddrNetmask)
with *Some* **show** *?thesis*
by(*simp add*: *ipassmt-iface-replace-dstip-mexpr-def bunch-of-lemmata-about-matches ipcidr-union-set-uncurry*)
qed

fun *oiface-rewrite*
:: *'i*::*len ipassignment* ⇒ *'i common-primitive match-expr* ⇒ *'i common-primitive match-expr*
where
oiface-rewrite - *MatchAny* = *MatchAny* |
oiface-rewrite ipassmt (*Match* (*OIface ifce*)) = *ipassmt-iface-replace-dstip-mexpr ipassmt ifce* |
oiface-rewrite - (*Match a*) = *Match a* |
oiface-rewrite ipassmt (*MatchNot m*) = *MatchNot* (*oiface-rewrite ipassmt m*) |
oiface-rewrite ipassmt (*MatchAnd m1 m2*) = *MatchAnd* (*oiface-rewrite ipassmt m1*) (*oiface-rewrite ipassmt m2*)

context
begin

private lemma *oiface-rewrite-matches-Primitive*:
matches (*common-matcher*, *α*) (*MatchNot* (*oiface-rewrite ipassmt* (*Match x*))) *a p* = *matches* (*common-matcher*, *α*) (*MatchNot* (*Match x*)) *a p* ⟷
matches (*common-matcher*, *α*) (*oiface-rewrite ipassmt* (*Match x*)) *a p* = *matches* (*common-matcher*, *α*) (*Match x*) *a p*
proof(*cases x*)
case (*OIface ifce*)
have (*matches* (*common-matcher*, *α*) (*MatchNot* (*ipassmt-iface-replace-dstip-mexpr ipassmt ifce*)) *a p* = (¬ *match-iface ifce* (*p-oiface p*))) ⟷
(*matches* (*common-matcher*, *α*) (*ipassmt-iface-replace-dstip-mexpr ipassmt ifce*) *a p* = *match-iface ifce* (*p-oiface p*))
proof(*cases ipassmt ifce*)
case *None* **thus** *?thesis*
apply(*simp add*: *matches-ipassmt-iface-replace-dstip-mexpr*)
apply(*simp add*: *ipassmt-iface-replace-dstip-mexpr-def primitive-matcher-generic.Iface-single-not*[*OF primitive-matcher-generic-common-matcher*])
done
next

case (*Some ips*)
{ fix *ips*
have *matches* (*common-matcher*, α)
(*MatchNot* (*match-list-to-match-expr* (*map* (*Match* ∘ *Dst* ∘ (*uncurry IpAddrNetmask*)) *ips*))) *a p* ⟷
(*p-dst p* ∉ *ipcidr-union-set* (*set ips*))
apply(*induction ips*)
apply(*simp add*: *bunch-of-lemmata-about-matches ipcidr-union-set-uncurry*)
apply(*simp add*: *MatchOr-MatchNot*)
apply(*simp add*: *ipcidr-union-set-uncurry*)
apply(*simp add*: *match-simplematcher-SrcDst-not*)
apply(*thin-tac* -)
apply(*simp add*: *ipt-iprange-to-set-uncurry-IpAddrNetmask*)
done
} note *helper*=*this*
from *Some* **show** *?thesis*
apply(*simp add*: *matches-ipassmt-iface-replace-dstip-mexpr*)
apply(*simp add*: *ipassmt-iface-replace-dstip-mexpr-def*)
apply(*simp add*: *helper*)
done
qed
with *OIface* **show** *?thesis*
by(*simp add*: *primitive-matcher-generic.Iface-single-not*[*OF primitive-matcher-generic-common-matcher*] *primitive-matcher-generic.Iface-single*[*OF primitive-matcher-generic-common-matcher*])
qed(*simp-all*)

lemma *ipassmt-disjoint-matcheq-iifce-dstip*:
assumes *ipassmt-nowild*: *ipassmt-sanity-nowildcards ipassmt*
and *ipassmt-disjoint*: *ipassmt-sanity-disjoint ipassmt*
and *ifce*: *ipassmt ifce* = *Some i-ips*
and *p-ifce*: *ipassmt* (*Iface* (*p-oiface p*)) = *Some p-ips* ∧ *p-dst p* ∈ *ipcidr-union-set* (*set p-ips*)
shows *match-iface ifce* (*p-oiface p*) ⟷ *p-dst p* ∈ *ipcidr-union-set* (*set i-ips*)
proof
assume *match-iface ifce* (*p-oiface p*)
thus *p-dst p* ∈ *ipcidr-union-set* (*set i-ips*)
apply(*cases ifce* = *Iface* (*p-oiface p*))
using *ifce p-ifce* **apply** *force*
by (*metis domI iface.sel iface-is-wildcard-def ifce ipassmt-nowild ipassmt-sanity-nowildcards-def match-iface.elims*(*2*) *match-iface-case-nowildcard*)
next
assume *a*: *p-dst p* ∈ *ipcidr-union-set* (*set i-ips*)
— basically, we need to reverse the map *ipassmt*

from *ipassmt-disjoint-nonempty-inj*[*OF ipassmt-disjoint ifce*] *a* **have** *ipassmt-inj*: ∀ *k*. *ipassmt k* = *Some i-ips* ⟶ *k* = *ifce* **by** *blast*

from *ipassmt-disjoint-inj-k*[*OF ipassmt-disjoint ifce* - *a*] **have** *ipassmt-inj-k*:

$\bigwedge k\ ips'.\ ipassmt\ k = Some\ ips' \Longrightarrow p\text{-}dst\ p \in ipcidr\text{-}union\text{-}set\ (set\ ips') \Longrightarrow k = ifce$ **by** *simp*

have *ipassmt-inj-p*: $\forall\ ips'.\ p\text{-}dst\ p \in ipcidr\text{-}union\text{-}set\ (set\ ips') \wedge (\exists\ k.\ ipassmt\ k = Some\ ips') \longrightarrow ips' = i\text{-}ips$

proof(*intro allI impI*; *elim conjE exE*)
fix *ips' k*
assume *as*: *p-dst p* $\in$ *ipcidr-union-set* (*set ips'*) *ipassmt k* = *Some ips'*
hence *k* = *ifce* **using** *ipassmt-inj-k* **by** *simp*
thus *ips'* = *i-ips* **using** *ifce as* **by** *simp*
qed

from *p-ifce* **have** (*Iface* (*p-oiface p*)) = *ifce* **using** *ipassmt-inj-p ipassmt-inj* **by** *blast*

thus *match-iface ifce* (*p-oiface p*) **using** *match-iface-refl* **by** *blast*
qed

private lemma *matches-ipassmt-iface-replace-dstip-mexpr-case-Iface*:
fixes *ifce*::*iface*
assumes *ipassmt-sanity-nowildcards ipassmt*
and *ipassmt-sanity-disjoint ipassmt*
and *ipassmt* (*Iface* (*p-oiface p*)) = *Some p-ips* $\wedge$ *p-dst p* $\in$ *ipcidr-union-set* (*set p-ips*)
shows *matches* (*common-matcher*, α) (*ipassmt-iface-replace-dstip-mexpr ipassmt ifce*) *a p* $\longleftrightarrow$
matches (*common-matcher*, α) (*Match* (*OIface ifce*)) *a p*
proof −
have *matches* (*common-matcher*, α) (*ipassmt-iface-replace-dstip-mexpr ipassmt ifce*) *a p* = *match-iface ifce* (*p-oiface p*)
proof −
show *?thesis*
proof(*cases ipassmt ifce*)
case *None* **thus** *?thesis* **by**(*simp add*: *matches-ipassmt-iface-replace-dstip-mexpr*)
next
case (*Some y*) **with** *assms*(*2*) **have** *p-dst p* $\in$ *ipcidr-union-set* (*set y*) = *match-iface ifce* (*p-oiface p*)
using *assms*(*1*) *assms*(*3*) *ipassmt-disjoint-matcheq-iifce-dstip* **by** *blast*
with *Some* **show** *?thesis* **by**(*simp add*: *matches-ipassmt-iface-replace-dstip-mexpr*)
qed
qed
thus *?thesis* **by**(*simp add*: *primitive-matcher-generic.Iface-single*[*OF primitive-matcher-generic-common-matcher*])
qed

```
  lemma matches-oiface-rewrite-ipassmt:
     normalized-nnf-match m ⟹ ipassmt-sanity-nowildcards ipassmt ⟹ ipassmt-sanity-disjoint
ipassmt ⟹
      (∃ p-ips. ipassmt (Iface (p-oiface p)) = Some p-ips ∧ p-dst p ∈ ipcidr-union-set
(set p-ips)) ⟹
        matches (common-matcher, α) (oiface-rewrite ipassmt m) a p ⟷ matches
(common-matcher, α) m a p
    proof(induction m)
    case MatchAny thus ?case by simp
    next
    case (MatchNot m)
      hence IH: normalized-nnf-match m ⟹
          matches (common-matcher, α) (oiface-rewrite ipassmt m) a p =matches
(common-matcher, α) m a p by blast
    with MatchNot.prems IH show ?case by(induction m) (simp-all add: oiface-rewrite-matches-Primitive)
    next
    case(Match x) thus ?case
      proof(cases x)
        case (OIface ifce) with Match show ?thesis
        apply(cases ipassmt (Iface (p-oiface p)))
         prefer 2
         apply(simp add: matches-ipassmt-iface-replace-dstip-mexpr-case-Iface; fail)
        by auto
      qed(simp-all)
    next
   case (MatchAnd m1 m2) thus ?case by(simp add: bunch-of-lemmata-about-matches)
    qed

  lemma matches-oiface-rewrite:
       normalized-nnf-match m ⟹ ipassmt-sanity-nowildcards ipassmt — TODO:
check? ⟹
        correct-routing rt ⟹
        ipassmt = map-of (routing-ipassmt rt) ⟹
        output-iface (routing-table-semantics rt (p-dst p)) = p-oiface p ⟹
        matches (common-matcher, α) (oiface-rewrite ipassmt m) a p ⟷ matches
(common-matcher, α) m a p
  apply(rule matches-oiface-rewrite-ipassmt; assumption?)
  apply(simp add: correct-routing-def routing-ipassmt-ipassmt-sanity-disjoint; fail)
  apply(simp)
  apply(rule routing-ipassmt; assumption?)
   apply(simp add: correct-routing-def; fail)
  done
end

lemma oiface-rewrite-preserves-nodisc:
  ∀ a. ¬ disc (Dst a) ⟹ ¬ has-disc disc m ⟹ ¬ has-disc disc (oiface-rewrite
ipassmt m)
  proof(induction ipassmt m rule: oiface-rewrite.induct)
```

case *2*
have $\forall$ *a*. $\neg$ *disc* (*Dst a*) $\Longrightarrow$ $\neg$ *disc* (*OIface ifce*) $\Longrightarrow$ $\neg$ *has-disc disc* (*ipassmt-iface-replace-dstip-mexpr ipassmt ifce*)
for *ifce ipassmt*
apply(*simp add*: *ipassmt-iface-replace-dstip-mexpr-def split*: *option.split*)
apply(*intro allI impI*, *rename-tac ips*)
apply(*drule-tac X=Dst* **and** *ls=map* (*uncurry IpAddrNetmask*) *ips* **in** *match-list-to-match-expr-not-has-dis*
apply(*simp*)
done
with *2* **show** *?case* **by** *simp*
qed(*simp-all*)

end
theory *Interface-Replace*
imports
No-Spoof
Common-Primitive-toString
Output-Interface-Replace
begin

36 Trying to connect inbound interfaces by their IP ranges

36.1 Constraining Interfaces

We keep the match on the interface but add the corresponding IP address range.

definition *ipassmt-iface-constrain-srcip-mexpr*
:: *'i*::*len ipassignment* $\Rightarrow$ *iface* $\Rightarrow$ *'i common-primitive match-expr*
where
ipassmt-iface-constrain-srcip-mexpr ipassmt ifce = (*case ipassmt ifce of*
None $\Rightarrow$ *Match* (*IIface ifce*)
| *Some ips* $\Rightarrow$ *MatchAnd*
(*Match* (*IIface ifce*))
(*match-list-to-match-expr* (*map* (*Match* $\circ$ *Src*) (*map* (*uncurry IpAddr-Netmask*) *ips*)))
)

lemma *matches-ipassmt-iface-constrain-srcip-mexpr*:
matches (*common-matcher*, α) (*ipassmt-iface-constrain-srcip-mexpr ipassmt ifce*) *a p* $\longleftrightarrow$
(*case ipassmt ifce of*
None $\Rightarrow$ *match-iface ifce* (*p-iiface p*)
| *Some ips* $\Rightarrow$ *match-iface ifce* (*p-iiface p*) $\wedge$ *p-src p* $\in$ *ipcidr-union-set* (*set ips*)
)
proof(*cases ipassmt ifce*)

case *None* **thus** *?thesis* **by**(*simp add*: *ipassmt-iface-constrain-srcip-mexpr-def primitive-matcher-generic.Iface-single*[*OF primitive-matcher-generic-common-matcher*]; *fail*)
next
case (*Some ips*)
have *matches* (*common-matcher*, α) (*match-list-to-match-expr* (*map* (*Match* $\circ$ *Src* $\circ$ (*uncurry IpAddrNetmask*)) *ips*)) *a p* $\longleftrightarrow$
(∃ *m*∈*set ips*. *p-src p* ∈ *uncurry ipset-from-cidr m*)
apply(*simp add*: *match-list-to-match-expr-disjunction*[*symmetric*]
match-list-matches match-simplematcher-SrcDst)
by(*simp add*: *ipt-iprange-to-set-uncurry-IpAddrNetmask*)
with *Some* **show** *?thesis*
apply(*simp add*: *ipcidr-union-set-uncurry*)
apply(*simp add*: *ipassmt-iface-constrain-srcip-mexpr-def bunch-of-lemmata-about-matches*)
apply(*simp add*: *primitive-matcher-generic.Iface-single*[*OF primitive-matcher-generic-common-matcher*])
done
qed

fun *iiface-constrain* :: *'i*::*len ipassignment* ⇒ *'i common-primitive match-expr* ⇒ *'i common-primitive match-expr* **where**
iiface-constrain - *MatchAny* = *MatchAny* |
iiface-constrain ipassmt (*Match* (*IIface ifce*)) = *ipassmt-iface-constrain-srcip-mexpr ipassmt ifce* |
iiface-constrain ipassmt (*Match a*) = *Match a* |
iiface-constrain ipassmt (*MatchNot m*) = *MatchNot* (*iiface-constrain ipassmt m*) |
iiface-constrain ipassmt (*MatchAnd m1 m2*) = *MatchAnd* (*iiface-constrain ipassmt m1*) (*iiface-constrain ipassmt m2*)

context
begin

private lemma *iiface-constrain-matches-Primitive*:
matches (*common-matcher*, α) (*MatchNot* (*iiface-constrain ipassmt* (*Match x*))) *a p* = *matches* (*common-matcher*, α) (*MatchNot* (*Match x*)) *a p* $\longleftrightarrow$
matches (*common-matcher*, α) (*iiface-constrain ipassmt* (*Match x*)) *a p* = *matches* (*common-matcher*, α) (*Match x*) *a p*
proof(*cases x*)
case (*IIface ifce*)
have (*matches* (*common-matcher*, α) (*MatchNot* (*ipassmt-iface-constrain-srcip-mexpr ipassmt ifce*)) *a p* = (¬ *match-iface ifce* (*p-iiface p*))) $\longleftrightarrow$
(*matches* (*common-matcher*, α) (*ipassmt-iface-constrain-srcip-mexpr ipassmt ifce*) *a p* = *match-iface ifce* (*p-iiface p*))
proof(*cases ipassmt ifce*)
case *None* **thus** *?thesis*
apply(*simp add*: *matches-ipassmt-iface-constrain-srcip-mexpr*)
apply(*simp add*: *ipassmt-iface-constrain-srcip-mexpr-def*

```
        primitive-matcher-generic.Iface-single-not[OF primitive-matcher-generic-common-matcher])
    done
  next
  case (Some ips)
    { fix ips
      have matches (common-matcher, α)
            (MatchNot (match-list-to-match-expr (map (Match ∘ Src ∘ (uncurry
IpAddrNetmask)) ips))) a p ⟷
            (p-src p ∉ ipcidr-union-set (set ips))
      apply(induction ips)
      apply(simp add: bunch-of-lemmata-about-matches ipcidr-union-set-uncurry;
fail)
      apply(simp add: MatchOr-MatchNot)
      apply(simp add: ipcidr-union-set-uncurry)
      apply(simp add: match-simplematcher-SrcDst-not)
      apply(thin-tac -)
      by (simp add: ipt-iprange-to-set-uncurry-IpAddrNetmask)
    } note helper=this
    from Some show ?thesis
      apply(simp add: matches-ipassmt-iface-constrain-srcip-mexpr)
    apply(simp add: ipassmt-iface-constrain-srcip-mexpr-def primitive-matcher-generic.Iface-single-not[OF
primitive-matcher-generic-common-matcher])
      apply(simp add: matches-DeMorgan)
      apply(simp add: helper)
        apply(simp add: primitive-matcher-generic.Iface-single-not[OF primi-
tive-matcher-generic-common-matcher])
      by blast
  qed
  with IIface show ?thesis
  by(simp add: primitive-matcher-generic.Iface-single-not[OF primitive-matcher-generic-common-matcher]
        primitive-matcher-generic.Iface-single[OF primitive-matcher-generic-common-matcher])
qed(simp-all)



private lemma matches-ipassmt-iface-constrain-srcip-mexpr-case-Iface:
    fixes ifce::iface
    assumes ipassmt-sanity-nowildcards ipassmt
      and ⋀ips. ipassmt (Iface (p-iiface p)) = Some ips ⟹ p-src p ∈ ip-
cidr-union-set (set ips)
    shows matches (common-matcher, α) (ipassmt-iface-constrain-srcip-mexpr
ipassmt ifce) a p ⟷
          matches (common-matcher, α) (Match (IIface ifce)) a p
proof −
  have matches (common-matcher, α) (ipassmt-iface-constrain-srcip-mexpr ipassmt
ifce) a p = match-iface ifce (p-iiface p)
    proof(cases ipassmt (Iface (p-iiface p)))
    case None
    from None show ?thesis
```

proof(*cases ipassmt ifce*)
case *None* **thus** *?thesis* **by**(*simp add*: *matches-ipassmt-iface-constrain-srcip-mexpr*)
next
case (*Some a*)
from *assms*(*1*) **have** ¬ *match-iface ifce* (*p-iiface p*)
apply(*rule ipassmt-sanity-nowildcards-match-iface*)
by(*simp-all add*: *Some None*)
with *Some* **show** *?thesis* **by**(*simp add*: *matches-ipassmt-iface-constrain-srcip-mexpr*)
qed
next
case (*Some x*)
with *assms*(*2*) **have** *assms2*: *p-src p* ∈ *ipcidr-union-set* (*set x*) **by**(*simp*)
show *?thesis*
proof(*cases ipassmt ifce*)
case *None* **thus** *?thesis* **by**(*simp add*: *matches-ipassmt-iface-constrain-srcip-mexpr*)
next
case (*Some y*) **with** *assms*(*2*) **have** (*match-iface ifce* (*p-iiface p*) ∧ *p-src p* ∈ *ipcidr-union-set* (*set y*)) = *match-iface ifce* (*p-iiface p*)
apply(*cases ifce*)
apply(*rename-tac ifce-str*)
apply(*case-tac ifce-str* = (*p-iiface p*))
apply (*simp add*: *match-iface-refl*; *fail*)
apply(*simp*)
apply(*subgoal-tac* ¬ *match-iface* (*Iface ifce-str*) (*p-iiface p*))
apply(*simp*)
using *assms*(*1*) **by** (*metis domI iface.sel iface-is-wildcard-def ipassmt-sanity-nowildcards-def match-iface-case-nowildcard*)
with *Some* **show** *?thesis* **by**(*simp add*: *matches-ipassmt-iface-constrain-srcip-mexpr*)
qed
qed
thus *?thesis* **by**(*simp add*: *primitive-matcher-generic.Iface-single*[*OF primitive-matcher-generic-common-matcher*])
qed

lemma *matches-iiface-constrain*:
normalized-nnf-match m ⟹ *ipassmt-sanity-nowildcards ipassmt* ⟹
(⋀*ips*. *ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* ⟹ *p-src p* ∈ *ipcidr-union-set* (*set ips*)) ⟹
matches (*common-matcher*, α) (*iiface-constrain ipassmt m*) *a p* ⟷ *matches* (*common-matcher*, α) *m a p*
proof(*induction m*)
case *MatchAny* **thus** *?case* **by** *simp*
next
case (*MatchNot m*)
hence *IH*: *normalized-nnf-match m* ⟹ *matches* (*common-matcher*, α) (*iiface-constrain ipassmt m*) *a p* = *matches* (*common-matcher*, α) *m a p* **by** *blast*
with *MatchNot.prems IH* **show** *?case* **by**(*induction m*) (*simp-all add*: *iiface-constrain-matches-Primitive*)
next

case(*Match x*) **thus** *?case*
proof(*cases x*)
case (*IIface ifce*) **with** *Match* **show** *?thesis*
using *matches-ipassmt-iface-constrain-srcip-mexpr-case-Iface* **by** *fastforce*
qed(*simp-all*)
next
case (*MatchAnd m1 m2*) **thus** *?case* **by**(*simp add*: *bunch-of-lemmata-about-matches*)
qed
end

36.2 Sanity checking the assumption

lemma (∃ *ips*. *ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* ∧ *p-src p* ∈ *ipcidr-union-set* (*set ips*)) ⟹
(*case ipassmt* (*Iface* (*p-iiface p*)) *of Some ips* ⇒ *p-src p* ∈ *ipcidr-union-set* (*set ips*))
(*case ipassmt* (*Iface* (*p-iiface p*)) *of Some ips* ⇒ *p-src p* ∈ *ipcidr-union-set* (*set ips*)) ⟹
(⋀*ips*. *ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* ⟹ *p-src p* ∈ *ipcidr-union-set* (*set ips*))
by(*cases ipassmt* (*Iface* (*p-iiface p*)),*simp-all*)+

Sanity check: If we assume that there are no spoofed packets, spoofing protection is trivially fulfilled.

lemma ∀ *p*:: (*'i*::*len*,*'pkt-ext*) *tagged-packet-scheme*.
Iface (*p-iiface p*) ∈ *dom ipassmt* ⟶ *p-src p* ∈ *ipcidr-union-set* (*set* (*the* (*ipassmt* (*Iface* (*p-iiface p*))))) ⟹
no-spoofing TYPE(*'pkt-ext*) *ipassmt rs*
apply(*simp add*: *no-spoofing-def*)
apply(*clarify*)
apply(*rename-tac iface ips p*)
apply(*thin-tac* -,-⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow*)
apply(*erule-tac x*=*p*(|*p-iiface* := *iface-sel iface*|) **in** *allE*)
apply(*auto*)
done

Sanity check: If the firewall features spoofing protection and we look at a packet which was allowed by the firewall. Then the packet's src ip must be according to ipassmt. (case Some) We don't case about packets from an interface which are not defined in ipassmt. (case None)

lemma
fixes *p* :: (*'i*::*len*,*'pkt-ext*) *tagged-packet-scheme*
shows *no-spoofing TYPE*(*'pkt-ext*) *ipassmt rs* ⟹
(*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ⟹
case ipassmt (*Iface* (*p-iiface p*)) *of Some ips* ⇒ *p-src p* ∈ *ipcidr-union-set* (*set ips*) | *None* ⇒ *True*
apply(*simp add*: *no-spoofing-def*)
apply(*case-tac Iface* (*p-iiface p*) ∈ *dom ipassmt*)

apply(*erule-tac x=Iface* (*p-iiface p*) **in** *ballE*)
apply(*simp-all*)
apply(*erule-tac x=p* **in** *allE*)
apply(*simp*)
apply *fastforce*
by (*simp add*: *domIff*)

36.3 Replacing Interfaces Completely

This is a stricter, true rewriting since it removes the interface match completely. However, it requires *ipassmt-sanity-disjoint*

thm *ipassmt-sanity-disjoint-def*

definition *ipassmt-iface-replace-srcip-mexpr*
:: *'i*::*len ipassignment* ⇒ *iface* ⇒ *'i common-primitive match-expr* **where**
ipassmt-iface-replace-srcip-mexpr ipassmt ifce ≡ *case ipassmt ifce of*
None ⇒ *Match* (*IIface ifce*)
| *Some ips* ⇒ (*match-list-to-match-expr* (*map* (*Match* ∘ *Src*) (*map* (*uncurry IpAddrNetmask*) *ips*)))

lemma *matches-ipassmt-iface-replace-srcip-mexpr*:
matches (*common-matcher*, *α*) (*ipassmt-iface-replace-srcip-mexpr ipassmt ifce*) *a p* ⟷ (*case ipassmt ifce of*
None ⇒ *match-iface ifce* (*p-iiface p*)
| *Some ips* ⇒ *p-src p* ∈ *ipcidr-union-set* (*set ips*)
)
proof(*cases ipassmt ifce*)
case *None* **thus** *?thesis* **by**(*simp add*: *ipassmt-iface-replace-srcip-mexpr-def primitive-matcher-generic.Iface-single*[*OF primitive-matcher-generic-common-matcher*])
next
case (*Some ips*)
have *matches* (*common-matcher*, *α*) (*match-list-to-match-expr* (*map* (*Match* ∘ *Src* ∘ (*uncurry IpAddrNetmask*)) *ips*)) *a p* ⟷
(∃ *m*∈*set ips*. *p-src p* ∈ (*uncurry ipset-from-cidr m*))
by(*simp add*: *match-list-to-match-expr-disjunction*[*symmetric*]
match-list-matches match-simplematcher-SrcDst ipt-iprange-to-set-uncurry-IpAddrNetmask)
with *Some* **show** *?thesis*
apply(*simp add*: *ipassmt-iface-replace-srcip-mexpr-def bunch-of-lemmata-about-matches*)
apply(*simp add*: *ipcidr-union-set-uncurry*)
done
qed

fun *iiface-rewrite*
:: *'i*::*len ipassignment* ⇒ *'i common-primitive match-expr* ⇒ *'i common-primitive match-expr*
where
iiface-rewrite - *MatchAny* = *MatchAny* |

iiface-rewrite ipassmt (*Match* (*IIface ifce*)) = *ipassmt-iface-replace-srcip-mexpr ipassmt ifce* |
iiface-rewrite ipassmt (*Match a*) = *Match a* |
iiface-rewrite ipassmt (*MatchNot m*) = *MatchNot* (*iiface-rewrite ipassmt m*) |
iiface-rewrite ipassmt (*MatchAnd m1 m2*) = *MatchAnd* (*iiface-rewrite ipassmt m1*) (*iiface-rewrite ipassmt m2*)

context
begin

private lemma *iiface-rewrite-matches-Primitive*:
matches (*common-matcher*, *α*) (*MatchNot* (*iiface-rewrite ipassmt* (*Match x*))) *a p* = *matches* (*common-matcher*, *α*) (*MatchNot* (*Match x*)) *a p* ⟷
matches (*common-matcher*, *α*) (*iiface-rewrite ipassmt* (*Match x*)) *a p* = *matches* (*common-matcher*, *α*) (*Match x*) *a p*
proof(*cases x*)
case (*IIface ifce*)
have (*matches* (*common-matcher*, *α*) (*MatchNot* (*ipassmt-iface-replace-srcip-mexpr ipassmt ifce*)) *a p* = (¬ *match-iface ifce* (*p-iiface p*))) ⟷
(*matches* (*common-matcher*, *α*) (*ipassmt-iface-replace-srcip-mexpr ipassmt ifce*) *a p* = *match-iface ifce* (*p-iiface p*))
proof(*cases ipassmt ifce*)
case *None* **thus** *?thesis*
apply(*simp add*: *matches-ipassmt-iface-replace-srcip-mexpr*)
apply(*simp add*: *ipassmt-iface-replace-srcip-mexpr-def primitive-matcher-generic.Iface-single-not*[*OF primitive-matcher-generic-common-matcher*])
done
next
case (*Some ips*)
{ fix *ips*
have *matches* (*common-matcher*, *α*)
(*MatchNot* (*match-list-to-match-expr* (*map* (*Match* ∘ *Src* ∘ (*uncurry IpAddrNetmask*)) *ips*))) *a p* ⟷
(*p-src p* ∉ *ipcidr-union-set* (*set ips*))
apply(*induction ips*)
apply(*simp add*: *bunch-of-lemmata-about-matches ipcidr-union-set-uncurry*)
apply(*simp add*: *MatchOr-MatchNot*)
apply(*simp add*: *ipcidr-union-set-uncurry*)
apply(*simp add*: *match-simplematcher-SrcDst-not*)
apply(*thin-tac* -)
apply(*simp add*: *ipt-iprange-to-set-uncurry-IpAddrNetmask*)
done
} **note** *helper*=*this*
from *Some* **show** *?thesis*
apply(*simp add*: *matches-ipassmt-iface-replace-srcip-mexpr*)
apply(*simp add*: *ipassmt-iface-replace-srcip-mexpr-def*)
apply(*simp add*: *helper*)
done

qed
with *IIface* **show** *?thesis*
by(*simp add*: *primitive-matcher-generic.Iface-single-not*[*OF primitive-matcher-generic-common-matcher*]
primitive-matcher-generic.Iface-single[*OF primitive-matcher-generic-common-matcher*])
qed(*simp-all*)

private lemma *matches-ipassmt-iface-replace-srcip-mexpr-case-Iface*:
fixes *ifce*::*iface*
assumes *ipassmt-sanity-nowildcards ipassmt*
and *ipassmt-sanity-disjoint ipassmt*
and *ipassmt* (*Iface* (*p-iiface p*)) = *Some p-ips* ∧ *p-src p* ∈ *ipcidr-union-set* (*set p-ips*)
shows *matches* (*common-matcher*, α) (*ipassmt-iface-replace-srcip-mexpr ipassmt ifce*) *a p* ⟷
matches (*common-matcher*, α) (*Match* (*IIface ifce*)) *a p*
proof −
have *matches* (*common-matcher*, α) (*ipassmt-iface-replace-srcip-mexpr ipassmt ifce*) *a p* = *match-iface ifce* (*p-iiface p*)
proof −
show *?thesis*
proof(*cases ipassmt ifce*)
case *None* **thus** *?thesis* **by**(*simp add*: *matches-ipassmt-iface-replace-srcip-mexpr*)
next
case (*Some y*) **with** *assms*(*2*) **have** *p-src p* ∈ *ipcidr-union-set* (*set y*) = *match-iface ifce* (*p-iiface p*)
using *assms*(*1*) *assms*(*3*) *ipassmt-disjoint-matcheq-iifce-srcip* **by** *blast*
with *Some* **show** *?thesis* **by**(*simp add*: *matches-ipassmt-iface-replace-srcip-mexpr*)
qed
qed
thus *?thesis* **by**(*simp add*: *primitive-matcher-generic.Iface-single*[*OF primitive-matcher-generic-common-matcher*])
qed

lemma *matches-iiface-rewrite*:
normalized-nnf-match m ⟹ *ipassmt-sanity-nowildcards ipassmt* ⟹ *ipassmt-sanity-disjoint ipassmt* ⟹
(∃ *p-ips*. *ipassmt* (*Iface* (*p-iiface p*)) = *Some p-ips* ∧ *p-src p* ∈ *ipcidr-union-set* (*set p-ips*)) ⟹
matches (*common-matcher*, α) (*iiface-rewrite ipassmt m*) *a p* ⟷ *matches* (*common-matcher*, α) *m a p*
proof(*induction m*)
case *MatchAny* **thus** *?case* **by** *simp*
next
case (*MatchNot m*)
hence *IH*: *normalized-nnf-match m* ⟹

```
        matches (common-matcher, α) (iiface-rewrite ipassmt m) a p =matches
(common-matcher, α) m a p by blast
    with MatchNot.prems IH show ?case by(induction m) (simp-all add: iiface-rewrite-matches-Primitive)
    next
    case(Match x) thus ?case
      proof(cases x)
        case (IIface ifce) with Match show ?thesis
        apply(cases ipassmt (Iface (p-iiface p)))
         prefer 2
         apply(simp add: matches-ipassmt-iface-replace-srcip-mexpr-case-Iface; fail)
        by auto
      qed(simp-all)
    next
   case (MatchAnd m1 m2) thus ?case by(simp add: bunch-of-lemmata-about-matches)
    qed

end
```

Finally, we show that *ipassmt-sanity-disjoint* is really needed.

```
  lemma iface-replace-needs-ipassmt-disjoint:
    assumes ipassmt-sanity-nowildcards ipassmt
    and iface-replace: ⋀ ifce p:: 'i::len tagged-packet.
          (matches (common-matcher, α) (ipassmt-iface-replace-srcip-mexpr ipassmt
ifce) a p ⟷ matches (common-matcher, α) (Match (IIface ifce)) a p)
    shows ipassmt-sanity-disjoint ipassmt
  unfolding ipassmt-sanity-disjoint-def
  proof(intro ballI impI)
    fix i1 i2
    assume i1 ∈ dom ipassmt and i2 ∈ dom ipassmt and i1 ≠ i2
   from ‹i1 ∈ dom ipassmt› obtain i1-ips where i1-ips: ipassmt i1 = Some i1-ips
by blast
   from ‹i2 ∈ dom ipassmt› obtain i2-ips where i2-ips: ipassmt i2 = Some i2-ips
by blast

    { fix p :: 'i tagged-packet
      from iface-replace[of i1 p(| p-iiface := iface-sel i2 |)] have
        (p-src p ∈ ipcidr-union-set (set i2-ips) ⟹ (p-src p ∈ ipcidr-union-set (set
i1-ips)) = match-iface i1 (iface-sel i2))
     apply(simp add: primitive-matcher-generic.Iface-single[OF primitive-matcher-generic-common-matcher]
‹i1 ∈ dom ipassmt›)
      apply(simp add: matches-ipassmt-iface-replace-srcip-mexpr i1-ips)
      done
      with ‹i1 ≠ i2› have ¬ (p-src p ∈ ipcidr-union-set (set i2-ips) ∧ (p-src p ∈
ipcidr-union-set (set i1-ips)))
       by (metis ‹i1 ∈ dom ipassmt› assms(1) iface.exhaust-sel iface-is-wildcard-def
ipassmt-sanity-nowildcards-def match-iface-case-nowildcard)
    }
     hence ¬ (src ∈ ipcidr-union-set (set i2-ips) ∧ (src ∈ ipcidr-union-set (set
i1-ips)))
```

for *src*
apply(*simp*)
by (*metis simple-packet.select-convs*(*3*))

thus *ipcidr-union-set* (*set* (*the* (*ipassmt i1*))) ∩ *ipcidr-union-set* (*set* (*the* (*ipassmt i2*))) = {}
apply(*simp add*: *i1-ips i2-ips*)
by *blast*
qed

end
theory *Optimizing*
imports *Semantics-Ternary*
begin

37 Optimizing

37.1 Removing Shadowed Rules

Note: there is no executable code for rmshadow at the moment

Assumes: *simple-ruleset*

fun *rmshadow* :: ('*a*, '*p*) *match-tac* ⇒ '*a rule list* ⇒ '*p set* ⇒ '*a rule list* **where**
rmshadow - [] - = [] |
rmshadow γ ((*Rule m a*)#*rs*) *P* = (*if* (∀ *p*∈*P*. ¬ *matches* γ *m a p*)
then
rmshadow γ *rs P*
else
(*Rule m a*) # (*rmshadow* γ *rs* {*p* ∈ *P*. ¬ *matches* γ *m a p*}))

37.1.1 Soundness

lemma *rmshadow-sound*:
simple-ruleset rs ⟹ *p* ∈ *P* ⟹ *approximating-bigstep-fun* γ *p* (*rmshadow* γ *rs P*) = *approximating-bigstep-fun* γ *p rs*
proof(*induction rs arbitrary*: *P*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
let *?fw*=*approximating-bigstep-fun* γ — firewall semantics
let *?rm*=*rmshadow* γ
let *?match*=*matches* γ (*get-match r*) (*get-action r*)
let *?set*={*p* ∈ *P*. ¬ *?match p*}
from *Cons.IH Cons.prems* **have** *IH*: *?fw p* (*?rm rs P*) = *?fw p rs* **by** (*simp add*: *simple-ruleset-def*)
from *Cons.IH*[*of ?set*] *Cons.prems* **have** *IH'*: *p* ∈ *?set* ⟹ *?fw p* (*?rm rs ?set*) = *?fw p rs* **by** (*simp add*: *simple-ruleset-def*)
from *Cons* **show** *?case*
proof(*cases* ∀ *p*∈*P*. ¬ *?match p*) — the if-condition of rmshadow

```
      case True
        from True have 1: ?rm (r#rs) P = ?rm rs P
          apply(cases r)
          apply(rename-tac m a)
          apply(clarify)
          apply(simp)
          done
        from True Cons.prems have ?fw p (r # rs) = ?fw p rs
          apply(cases r)
          apply(rename-tac m a)
          apply(simp add: fun-eq-iff)
          apply(clarify)
       apply(rule just-show-all-approximating-bigstep-fun-equalities-with-start-Undecided)
          apply(simp)
          done
        from this IH have ?fw p (?rm rs P) = ?fw p (r#rs)  by simp
        thus ?fw p (?rm (r#rs) P) = ?fw p (r#rs)  using 1 by simp
      next
      case False — else
        have ?fw p (r # (?rm rs ?set)) = ?fw p (r # rs)
          proof(cases p ∈ ?set)
            case True
              from True IH' show ?fw p (r # (?rm rs ?set)) = ?fw p (r#rs)
                apply(cases r)
                apply(rename-tac m a)
                apply(simp add: fun-eq-iff)
                apply(clarify)
             apply(rule just-show-all-approximating-bigstep-fun-equalities-with-start-Undecided)
                apply(simp)
                done
             next
             case False
               from False Cons.prems have ?match p by simp
                from Cons.prems have get-action r = Accept ∨ get-action r = Drop
by(simp add: simple-ruleset-def)
              from this ‹?match p›show ?fw p (r # (?rm rs ?set)) = ?fw p (r#rs)
                apply(cases r)
                apply(rename-tac m a)
                apply(simp add: fun-eq-iff)
                apply(clarify)
                apply(rename-tac s)
             apply(rule just-show-all-approximating-bigstep-fun-equalities-with-start-Undecided)
                apply(simp split:action.split)
                apply fast
                done
          qed
        from False this show ?thesis
          apply(cases r)
          apply(rename-tac m a)
```

```
      apply(simp add: fun-eq-iff)
      apply(clarify)
    apply(rule just-show-all-approximating-bigstep-fun-equalities-with-start-Undecided)
      apply(simp)
      done
  qed
qed
```

37.2 Removing rules which cannot apply

```
fun rmMatchFalse :: 'a rule list ⇒ 'a rule list where
  rmMatchFalse [] = [] |
  rmMatchFalse ((Rule (MatchNot MatchAny) -)#rs) = rmMatchFalse rs |
  rmMatchFalse (r#rs) = r # rmMatchFalse rs

lemma rmMatchFalse-correct: approximating-bigstep-fun γ p (rmMatchFalse rs) s
= approximating-bigstep-fun γ p rs s
  proof−
    { fix m::'a match-expr and a and rs
      assume assm: m ≠ MatchNot MatchAny
      have rmMatchFalse (Rule m a # rs) = Rule m a # (rmMatchFalse rs) (is
?hlp)
      proof(cases m)
        case (MatchNot mexpr) with assm show ?hlp by(cases mexpr) simp-all
        qed(simp-all)
    } note rmMatchFalse-helper=this
  show ?thesis
    proof(induction γ p rs s rule: approximating-bigstep-fun-induct)
      case Empty thus ?case by(simp)
      next
      case Decision thus ?case by(metis Decision-approximating-bigstep-fun)
      next
      case (Nomatch γ p m a) thus ?case
        by(cases m = MatchNot MatchAny) (simp-all add: rmMatchFalse-helper)
      next
      case (Match γ p m a rs)
      from Match(1) have m ≠ MatchNot MatchAny using bunch-of-lemmata-about-matches(3)
by fast
        with Match rmMatchFalse-helper show ?case by(simp split:action.split)
    qed
  qed
```

We can stop after a default rule (a rule which matches anything) is observed.

```
fun cut-off-after-match-any :: 'a rule list ⇒ 'a rule list where
  cut-off-after-match-any [] = [] |
  cut-off-after-match-any (Rule m a # rs) =
    (if m = MatchAny ∧ (a = Accept ∨ a = Drop ∨ a = Reject)
     then [Rule m a] else Rule m a # cut-off-after-match-any rs)
```

lemma *cut-off-after-match-any*:
approximating-bigstep-fun γ *p* (*cut-off-after-match-any rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
apply(*rule just-show-all-approximating-bigstep-fun-equalities-with-start-Undecided*)
apply(*induction* γ *p rs s rule*: *approximating-bigstep-fun.induct*)
apply(*simp*; *fail*)
apply(*simp*; *fail*)
by(*simp split*: *action.split action.split-asm add*: *bunch-of-lemmata-about-matches*(*2*))

lemma *cut-off-after-match-any-simplers*: *simple-ruleset rs* $\Longrightarrow$ *simple-ruleset* (*cut-off-after-match-any rs*)
by(*induction rs rule*: *cut-off-after-match-any.induct*) (*simp-all add*: *simple-ruleset-def*)

lemma *cut-off-after-match-any-preserve-matches*:
$\forall$ *r* $\in$ *set rs*. *P* (*get-match r*) $\Longrightarrow$ $\forall$ *r* $\in$ *set* (*cut-off-after-match-any rs*). *P* (*get-match r*)
apply(*induction rs rule*: *cut-off-after-match-any.induct*)
apply(*simp*; *fail*)
by(*auto simp add*: *simple-ruleset-def*)

end

38 Optimizing and Normalizing Primitives

theory *Transform*
imports *Common-Primitive-Lemmas*
../Semantics-Ternary/Semantics-Ternary
../Semantics-Ternary/Negation-Type-Matching
Ports-Normalize
IpAddresses-Normalize
Interfaces-Normalize
Protocols-Normalize
../Common/Remdups-Rev
Interface-Replace
../Semantics-Ternary/Optimizing
begin

This transform theory plugs a lot of stuff together. We perform several normalization and optimization steps on complete firewall rulesets. We show that it preserves the semantics and also, that structural properties are preserved. For example, if you normalize interfaces and afterwards normalize protocols, the interfaces remain normalized and no new interfaces are added when doing the protocol normalization.

definition *compress-normalize-besteffort*
:: *'i*::*len common-primitive match-expr* $\Rightarrow$ *'i common-primitive match-expr option*
where
compress-normalize-besteffort m $\equiv$ *compress-normalize-primitive-monad*
[*compress-normalize-protocols*,

compress-normalize-input-interfaces,
compress-normalize-output-interfaces] *m*

context begin
private lemma *compress-normalize-besteffort-normalized*:
f ∈ *set* [*compress-normalize-protocols*,
compress-normalize-input-interfaces,
compress-normalize-output-interfaces] ⟹
normalized-nnf-match m ⟹ *f m* = *Some m′* ⟹ *normalized-nnf-match m′*
apply(*simp*)
apply(*elim disjE*)
using *compress-normalize-protocols-nnf* **apply** *blast*
using *compress-normalize-input-interfaces-nnf* **apply** *blast*
using *compress-normalize-output-interfaces-nnf* **apply** *blast*
done
private lemma *compress-normalize-besteffort-matches*:
assumes *generic*: *primitive-matcher-generic* β
shows *f* ∈ *set* [*compress-normalize-protocols*,
compress-normalize-input-interfaces,
compress-normalize-output-interfaces] ⟹
normalized-nnf-match m ⟹
f m = *Some m′* ⟹
matches (β, α) *m′ a p* = *matches* (β, α) *m a p*
apply(*simp*)
apply(*elim disjE*)
using *primitive-matcher-generic.compress-normalize-protocols-Some*[*OF generic*]
apply *blast*
using *compress-normalize-input-interfaces-Some*[*OF generic*] **apply** *blast*
using *compress-normalize-output-interfaces-Some*[*OF generic*] **apply** *blast*
done

lemma *compress-normalize-besteffort-Some*:
assumes *generic*: *primitive-matcher-generic* β
shows *normalized-nnf-match m* ⟹
compress-normalize-besteffort m = *Some m′* ⟹
matches (β, α) *m′ a p* = *matches* (β, α) *m a p*
unfolding *compress-normalize-besteffort-def*
apply(*rule compress-normalize-primitive-monad*)
using *compress-normalize-besteffort-normalized compress-normalize-besteffort-matches*[*OF generic*] **by** *blast+*
lemma *compress-normalize-besteffort-None*:
assumes *generic*: *primitive-matcher-generic* β
shows *normalized-nnf-match m* ⟹
compress-normalize-besteffort m = *None* ⟹
¬ *matches* (β, α) *m a p*
proof −
have *notmatches*: *f* ∈ *set* [*compress-normalize-protocols*, *compress-normalize-input-interfaces*, *compress-normalize-output-interfaces*] ⟹

```
        normalized-nnf-match m ⟹ f m = None ⟹ ¬ matches (β, α) m a p
for f m
    apply(simp)
   using primitive-matcher-generic.compress-normalize-protocols-None[OF generic]
         compress-normalize-input-interfaces-None[OF generic]
         compress-normalize-output-interfaces-None[OF generic] by blast
  show normalized-nnf-match m ⟹ compress-normalize-besteffort m = None ⟹
¬ matches (β, α) m a p
    unfolding compress-normalize-besteffort-def
    apply(rule compress-normalize-primitive-monad-None)
      using compress-normalize-besteffort-normalized
            compress-normalize-besteffort-matches[OF generic]
            notmatches by blast+
  qed
  lemma compress-normalize-besteffort-nnf:
    normalized-nnf-match m ⟹
     compress-normalize-besteffort m = Some m' ⟹
     normalized-nnf-match m'
    unfolding compress-normalize-besteffort-def
    apply(rule compress-normalize-primitive-monad)
      using compress-normalize-besteffort-normalized
        compress-normalize-besteffort-matches[OF primitive-matcher-generic-common-matcher]
          by blast+

  lemma compress-normalize-besteffort-not-introduces-Iiface:
    ¬ has-disc is-Iiface m ⟹ normalized-nnf-match m ⟹ compress-normalize-besteffort
m = Some m' ⟹
      ¬ has-disc is-Iiface m'
    unfolding compress-normalize-besteffort-def
    apply(rule compress-normalize-primitive-monad-preserves[THEN conjunct2])
     apply(drule(3) compress-normalize-besteffort-normalized)
      apply(auto dest: compress-normalize-input-interfaces-not-introduces-Iiface
                       compress-normalize-protocols-hasdisc
                       compress-normalize-output-interfaces-hasdisc)
    done
  lemma compress-normalize-besteffort-not-introduces-Oiface:
    ¬ has-disc is-Oiface m ⟹ normalized-nnf-match m ⟹ compress-normalize-besteffort
m = Some m' ⟹
      ¬ has-disc is-Oiface m'
    unfolding compress-normalize-besteffort-def
    apply(rule compress-normalize-primitive-monad-preserves[THEN conjunct2])
     apply(drule(3) compress-normalize-besteffort-normalized)
      apply(auto dest: compress-normalize-output-interfaces-hasdisc
                       compress-normalize-output-interfaces-not-introduces-Oiface
                       compress-normalize-protocols-hasdisc
                       compress-normalize-input-interfaces-hasdisc)
    done

  lemma compress-normalize-besteffort-not-introduces-Iiface-negated:
```

¬ *has-disc-negated is-Iiface False m* ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-besteffort m* = *Some m′* ⟹
¬ *has-disc-negated is-Iiface False m′*
unfolding *compress-normalize-besteffort-def*
apply(*rule compress-normalize-primitive-monad-preserves*[*THEN conjunct2*])
apply(*drule*(*3*) *compress-normalize-besteffort-normalized*)
apply(*auto dest*: *compress-normalize-besteffort-normalized compress-normalize-input-interfaces-not-introd*
compress-normalize-protocols-hasdisc-negated
compress-normalize-output-interfaces-hasdisc-negated)
done
lemma *compress-normalize-besteffort-not-introduces-Oiface-negated*:
¬ *has-disc-negated is-Oiface False m* ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-besteffort m* = *Some m′* ⟹
¬ *has-disc-negated is-Oiface False m′*
unfolding *compress-normalize-besteffort-def*
apply(*rule compress-normalize-primitive-monad-preserves*[*THEN conjunct2*])
apply(*drule*(*3*) *compress-normalize-besteffort-normalized*)
apply(*auto dest*: *compress-normalize-output-interfaces-not-introduces-Oiface-negated*
compress-normalize-input-interfaces-hasdisc-negated
compress-normalize-protocols-hasdisc-negated)
done
lemma *compress-normalize-besteffort-not-introduces-Prot-negated*:
¬ *has-disc-negated is-Prot False m* ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-besteffort m* = *Some m′* ⟹
¬ *has-disc-negated is-Prot False m′*
unfolding *compress-normalize-besteffort-def*
apply(*rule compress-normalize-primitive-monad-preserves*[*THEN conjunct2*])
apply(*drule*(*3*) *compress-normalize-besteffort-normalized*)
apply(*auto dest*: *compress-normalize-input-interfaces-hasdisc-negated*
compress-normalize-protocols-not-introduces-Prot-negated
compress-normalize-output-interfaces-hasdisc-negated)
done
lemma *compress-normalize-besteffort-hasdisc*:
¬ *has-disc disc m* ⟹ (∀ *a*. ¬ *disc* (*IIface a*)) ⟹ (∀ *a*. ¬ *disc* (*OIface a*)) ⟹ (∀ *a*. ¬ *disc* (*Prot a*)) ⟹
normalized-nnf-match m ⟹ *compress-normalize-besteffort m* = *Some m′* ⟹
normalized-nnf-match m′ ∧ ¬ *has-disc disc m′*
unfolding *compress-normalize-besteffort-def*
apply(*rule compress-normalize-primitive-monad-preserves*)
apply(*drule*(*3*) *compress-normalize-besteffort-normalized*)
apply(*auto dest*: *compress-normalize-input-interfaces-hasdisc*
compress-normalize-output-interfaces-hasdisc
compress-normalize-protocols-hasdisc)
done
lemma *compress-normalize-besteffort-hasdisc-negated*:
¬ *has-disc-negated disc False m* ⟹
(∀ *a*. ¬ *disc* (*IIface a*)) ⟹ (∀ *a*. ¬ *disc* (*OIface a*)) ⟹ (∀ *a*. ¬ *disc* (*Prot a*)) ⟹

normalized-nnf-match m ⟹ *compress-normalize-besteffort m* = *Some m′*
⟹
normalized-nnf-match m′ ∧ ¬ *has-disc-negated disc False m′*

unfolding *compress-normalize-besteffort-def*
apply(*rule compress-normalize-primitive-monad-preserves*)
apply(*drule*(*3*) *compress-normalize-besteffort-normalized*)
apply(*simp split*: *option.split-asm*)
using *compress-normalize-input-interfaces-hasdisc-negated*
compress-normalize-output-interfaces-hasdisc-negated
compress-normalize-protocols-hasdisc-negated **apply** *blast*
apply *simp-all*
done
lemma *compress-normalize-besteffort-preserves-normalized-n-primitive*:
normalized-n-primitive (*disc*, *sel*) *P m* ⟹
(∀ *a*. ¬ *disc* (*IIface a*)) ⟹ (∀ *a*. ¬ *disc* (*OIface a*)) ⟹ (∀ *a*. ¬ *disc* (*Prot a*))
⟹
normalized-nnf-match m ⟹ *compress-normalize-besteffort m* = *Some m′* ⟹
normalized-nnf-match m′ ∧ *normalized-n-primitive* (*disc*, *sel*) *P m′*
unfolding *compress-normalize-besteffort-def*
apply(*rule compress-normalize-primitive-monad-preserves*)
apply(*drule*(*3*) *compress-normalize-besteffort-normalized*)
apply(*auto dest*: *compress-normalize-input-interfaces-preserves-normalized-n-primitive*
compress-normalize-output-interfaces-preserves-normalized-n-primitive
compress-normalize-protocols-preserves-normalized-n-primitive)
done
end

39 Transforming rulesets

39.1 Optimizations

lemma *approximating-bigstep-fun-remdups-rev*:
approximating-bigstep-fun γ *p* (*remdups-rev rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof(*induction* γ *p rs s rule*: *approximating-bigstep-fun.induct*)
case *1* **thus** *?case* **by**(*simp add*: *remdups-rev-def*)
next
case *2* **thus** *?case* **by** (*simp add*: *Decision-approximating-bigstep-fun*)
next
case (*3* γ *p m a rs*) **thus** *?case*
proof(*cases matches* γ *m a p*)
case *False* **with** *3* **show** *?thesis*
by(*simp add*: *remdups-rev-fst remdups-rev-removeAll not-matches-removeAll*)

next
case *True*
{ fix *rs s*
have *approximating-bigstep-fun* γ *p* (*filter* ((≠) (*Rule m Log*)) *rs*) *s* =

approximating-bigstep-fun γ *p rs s*
proof(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)
qed(*auto simp add*: *Decision-approximating-bigstep-fun split*: *action.split*)
} **note** *helper-Log=this*
{ **fix** *rs s*
have *approximating-bigstep-fun* γ *p* (*filter* ((≠) (*Rule m Empty*)) *rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)
qed(*auto simp add*: *Decision-approximating-bigstep-fun split*: *action.split*)
} **note** *helper-Empty=this*
from *True 3* **show** *?thesis*
apply(*simp add*: *remdups-rev-fst split*: *action.split*)
apply(*safe*)
apply(*simp-all*)
apply(*simp-all add*: *remdups-rev-removeAll*)
apply(*simp-all add*: *removeAll-filter-not-eq helper-Empty helper-Log*)
done
qed
qed

lemma *remdups-rev-simplers*: *simple-ruleset rs* ⟹ *simple-ruleset* (*remdups-rev rs*)
by(*induction rs*) (*simp-all add*: *remdups-rev-def simple-ruleset-def*)

lemma *remdups-rev-preserve-matches*:
∀ *r*∈*set rs*. *P* (*get-match r*) ⟹ ∀ *r*∈*set* (*remdups-rev rs*). *P* (*get-match r*)
by(*induction rs*) (*simp-all add*: *remdups-rev-def simple-ruleset-def*)

39.2 Optimize and Normalize to NNF form

definition *transform-optimize-dnf-strict* :: *'i*::*len common-primitive rule list* ⇒ *'i common-primitive rule list* **where**
transform-optimize-dnf-strict = *cut-off-after-match-any* ∘
(*optimize-matches opt-MatchAny-match-expr* ∘
normalize-rules-dnf ∘ (*optimize-matches* (*opt-MatchAny-match-expr* ∘ *optimize-primitive-univ*)))

theorem *transform-optimize-dnf-strict-structure*:
assumes *simplers*: *simple-ruleset rs* **and** *wf*α: *wf-unknown-match-tac* α
shows *simple-ruleset* (*transform-optimize-dnf-strict rs*)
and ∀ *r* ∈ *set rs*. ¬ *has-disc disc* (*get-match r*) ⟹
∀ *r* ∈ *set* (*transform-optimize-dnf-strict rs*). ¬ *has-disc disc* (*get-match r*)
and ∀ *r* ∈ *set* (*transform-optimize-dnf-strict rs*). *normalized-nnf-match* (*get-match r*)
and ∀ *r* ∈ *set rs*. *normalized-n-primitive disc-sel f* (*get-match r*) ⟹
∀ *r* ∈ *set* (*transform-optimize-dnf-strict rs*). *normalized-n-primitive disc-sel f* (*get-match r*)
and ∀ *r* ∈ *set rs*. ¬ *has-disc-negated disc neg* (*get-match r*) ⟹

```
        ∀ r ∈ set (transform-optimize-dnf-strict rs). ¬ has-disc-negated disc neg
(get-match r)
  proof −
    show simplers-transform: simple-ruleset (transform-optimize-dnf-strict rs)
      unfolding transform-optimize-dnf-strict-def
      using simplers by (simp add: cut-off-after-match-any-simplers
          optimize-matches-simple-ruleset simple-ruleset-normalize-rules-dnf)

    define transform-optimize-dnf-strict-inner
      where transform-optimize-dnf-strict-inner =
      optimize-matches (opt-MatchAny-match-expr :: 'a common-primitive match-expr
⇒ 'a common-primitive match-expr) ∘
            normalize-rules-dnf ∘ (optimize-matches (opt-MatchAny-match-expr ∘
optimize-primitive-univ))
     have inner-outer: transform-optimize-dnf-strict = (cut-off-after-match-any ∘
transform-optimize-dnf-strict-inner)
    by(auto simp add: transform-optimize-dnf-strict-def transform-optimize-dnf-strict-inner-def)
    have tf1: transform-optimize-dnf-strict-inner (r#rs) =
    (optimize-matches opt-MatchAny-match-expr (normalize-rules-dnf (optimize-matches
(opt-MatchAny-match-expr ∘ optimize-primitive-univ) [r])))@
        transform-optimize-dnf-strict-inner rs for r rs
      unfolding transform-optimize-dnf-strict-inner-def
      apply(simp)
      apply(subst optimize-matches-fst)
      apply(simp add: normalize-rules-dnf-append optimize-matches-append)
      done

    — if the individual optimization functions preserve a property, then the whole
thing does
    { fix P :: 'a::len common-primitive match-expr ⇒ bool
      assume p1: ∀ m. P m ⟶ P (optimize-primitive-univ m)
      assume p2: ∀ m. P m ⟶ P (opt-MatchAny-match-expr m)
      assume p3: ∀ m. P m ⟶ (∀ m' ∈ set (normalize-match m). P m')
      { fix rs
        have ∀ r ∈ set rs. P (get-match r) ⟹
       ∀ r ∈ set (optimize-matches (opt-MatchAny-match-expr ∘ optimize-primitive-univ)
rs). P (get-match r)
          apply(rule optimize-matches-preserves)
          using p1 p2 by simp
      } note opt1=this
      { fix rs
        have ∀ r ∈ set rs. P (get-match r) ⟹ ∀ r ∈ set (normalize-rules-dnf rs).
P (get-match r)
        apply(induction rs rule: normalize-rules-dnf.induct)
         apply(simp; fail)
        apply(simp)
        apply(safe)
         apply(simp-all)
        using p3 by(simp)
```

```
    } note opt2=this
    { fix rs
      have ∀ r ∈ set rs. P (get-match r) ⟹
        ∀ r ∈ set (optimize-matches opt-MatchAny-match-expr rs). P (get-match
r)
        apply(rule optimize-matches-preserves)
        using p2 by simp
    } note opt3=this
    have ∀ r ∈ set rs. P (get-match r) ⟹
      ∀ r ∈ set (transform-optimize-dnf-strict rs). P (get-match r)
      unfolding transform-optimize-dnf-strict-def
      apply(drule opt1)
      apply(drule opt2)
      apply(drule opt3)
      using cut-off-after-match-any-preserve-matches by auto
  } note matchpred-rule=this

  { fix m
    have ¬ has-disc disc m ⟹ ¬ has-disc disc (optimize-primitive-univ m)
    by(induction m rule: optimize-primitive-univ.induct) (simp-all)
  } moreover { fix m
   have ¬ has-disc disc m ⟶ (∀ m' ∈ set (normalize-match m). ¬ has-disc disc
m')
      using normalize-match-preserves-nodisc by fast
  } ultimately show ∀ r ∈ set rs. ¬ has-disc disc (get-match r) ⟹
    ∀ r ∈ set (transform-optimize-dnf-strict rs). ¬ has-disc disc (get-match r)
  using not-has-disc-opt-MatchAny-match-expr matchpred-rule[of λm. ¬ has-disc
disc m] by fast

  { fix m
  have ¬ has-disc-negated disc neg m ⟹ ¬ has-disc-negated disc neg (optimize-primitive-univ
m)
    apply(induction disc neg m rule: has-disc-negated.induct)
        apply(simp-all)
    apply(rename-tac a)
      apply(subgoal-tac optimize-primitive-univ (Match a) = Match a ∨ opti-
mize-primitive-univ (Match a) = MatchAny)
     apply safe
      apply simp-all
    using optimize-primitive-univ-unchanged-primitives by blast
  } with not-has-disc-negated-opt-MatchAny-match-expr not-has-disc-normalize-match
show
    ∀ r ∈ set rs. ¬ has-disc-negated disc neg (get-match r) ⟹
      ∀ r ∈ set (transform-optimize-dnf-strict rs). ¬ has-disc-negated disc neg
(get-match r)
    using matchpred-rule[of λm. ¬ has-disc-negated disc neg m] by fast

  { fix P and a::'a common-primitive
  have (optimize-primitive-univ (Match a)) = (Match a) ∨ (optimize-primitive-univ
```

(*Match a*)) = *MatchAny*
by(*induction* (*Match a*) *rule*: *optimize-primitive-univ.induct*) (*auto*)
hence ((*optimize-primitive-univ* (*Match a*)) = *Match a* ⟹ *P a*) ⟹ (*optimize-primitive-univ* (*Match a*) = *MatchAny* ⟹ *P a*) ⟹ *P a* **by** *blast*
} note *optimize-primitive-univ-match-cases=this*

{ fix *m*
have *normalized-n-primitive disc-sel f m* ⟹ *normalized-n-primitive disc-sel f* (*optimize-primitive-univ m*)
apply(*induction disc-sel f m rule*: *normalized-n-primitive.induct*)
apply(*simp-all split*: *if-split-asm*)
apply(*rule optimize-primitive-univ-match-cases*, *simp-all*)+
done
} moreover { fix *m*
have *normalized-n-primitive disc-sel f m* ⟶ (∀ *m′* ∈ *set* (*normalize-match m*). *normalized-n-primitive disc-sel f m′*)
using *normalize-match-preserves-normalized-n-primitive* **by** *blast*
} ultimately show ∀ *r* ∈ *set rs*. *normalized-n-primitive disc-sel f* (*get-match r*) ⟹
∀ *r* ∈ *set* (*transform-optimize-dnf-strict rs*). *normalized-n-primitive disc-sel f* (*get-match r*)
using *matchpred-rule*[*of* λ*m*. *normalized-n-primitive disc-sel f m*] *normalized-n-primitive-opt-MatchAny-match-expr* **by** *fast*

{ fix *rs*::*′a*::*len common-primitive rule list*
from *normalize-rules-dnf-normalized-nnf-match*[*of rs*]
have ∀ *x* ∈ *set* (*normalize-rules-dnf rs*). *normalized-nnf-match* (*get-match x*)
.
hence ∀ *r* ∈ *set* (*optimize-matches opt-MatchAny-match-expr* (*normalize-rules-dnf rs*)). *normalized-nnf-match* (*get-match r*)
apply −
apply(*rule optimize-matches-preserves*)
using *normalized-nnf-match-opt-MatchAny-match-expr* **by** *blast*
}
from *this* **have** ∀ *r* ∈ *set* (*transform-optimize-dnf-strict-inner rs*). *normalized-nnf-match* (*get-match r*)
unfolding *transform-optimize-dnf-strict-inner-def* **by** *simp*
thus ∀ *r* ∈ *set* (*transform-optimize-dnf-strict rs*). *normalized-nnf-match* (*get-match r*)
unfolding *inner-outer*
apply *simp*
apply(*rule cut-off-after-match-any-preserve-matches*)
.
qed

theorem *transform-optimize-dnf-strict*:
assumes *simplers*: *simple-ruleset rs* **and** *wf*α: *wf-unknown-match-tac* α
shows (*common-matcher*, α),*p*⊢ ⟨*transform-optimize-dnf-strict rs*, *s*⟩ ⇒α *t* ⟷

(*common-matcher*, α),*p*⊢ $\langle rs, s\rangle \Rightarrow_\alpha t$
proof −
let *?γ*=(*common-matcher*, α)
let *?fw*=λ*rs. approximating-bigstep-fun ?γ p rs s*

have *simplers-transform*: *simple-ruleset* (*transform-optimize-dnf-strict rs*)
unfolding *transform-optimize-dnf-strict-def*
using *simplers* **by** (*simp add*: *cut-off-after-match-any-simplers optimize-matches-simple-ruleset simple-ruleset-normalize-rules-dnf*)

have *simplers1*: *simple-ruleset* (*optimize-matches* (*opt-MatchAny-match-expr* ∘ *optimize-primitive-univ*) *rs*)
using *simplers optimize-matches-simple-ruleset* **by** (*metis*)

have *1*: *?γ*,*p*⊢ $\langle rs, s\rangle \Rightarrow_\alpha t \longleftrightarrow$ *?fw rs* = *t*
using *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers*]] **by** *fast*

have *?fw rs* = *?fw* (*optimize-matches* (*opt-MatchAny-match-expr* ∘ *optimize-primitive-univ*) *rs*)
apply(*rule optimize-matches*[*symmetric*])
using *optimize-primitive-univ-correct-matchexpr opt-MatchAny-match-expr-correct* **by** (*metis comp-apply*)
also have . . . = *?fw* (*normalize-rules-dnf* (*optimize-matches* (*opt-MatchAny-match-expr* ∘ *optimize-primitive-univ*) *rs*))
apply(*rule normalize-rules-dnf-correct*[*symmetric*])
using *simplers1* **by** (*metis good-imp-wf-ruleset simple-imp-good-ruleset*)
also have . . . = *?fw* (*optimize-matches opt-MatchAny-match-expr* (*normalize-rules-dnf* (*optimize-matches* (*opt-MatchAny-match-expr* ∘ *optimize-primitive-univ*) *rs*)))
apply(*rule optimize-matches*[*symmetric*])
using *opt-MatchAny-match-expr-correct* **by** (*metis*)
finally have *rs*: *?fw rs* = *?fw* (*transform-optimize-dnf-strict rs*)
unfolding *transform-optimize-dnf-strict-def* **by**(*simp add*: *cut-off-after-match-any*)

have *2*: *?fw* (*transform-optimize-dnf-strict rs*) = *t* $\longleftrightarrow$ *?γ*,*p*⊢ $\langle$*transform-optimize-dnf-strict rs*, *s*$\rangle \Rightarrow_\alpha t$
using *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers-transform*], *symmetric*] **by** *fast*
from *1 2 rs* **show** *?γ*,*p*⊢ $\langle$*transform-optimize-dnf-strict rs*, *s*$\rangle \Rightarrow_\alpha t \longleftrightarrow$ *?γ*,*p*⊢ $\langle rs, s\rangle \Rightarrow_\alpha t$ **by** *simp*
qed

39.3 Abstracting over unknowns

definition *transform-remove-unknowns-generic*
:: (*'a*, *'packet*) *match-tac* ⇒ *'a rule list* ⇒ *'a rule list*
where
transform-remove-unknowns-generic γ = *optimize-matches-a* (*remove-unknowns-generic*

γ)

theorem *transform-remove-unknowns-generic*:
assumes *simplers*: *simple-ruleset rs*
and *wf*α: *wf-unknown-match-tac* α **and** *packet-independent-*α: *packet-independent-*α α
and *wf*β: *packet-independent-*β*-unknown* β
shows (β, α),*p*$\vdash$ $\langle$*transform-remove-unknowns-generic* (β, α) *rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t* $\longleftrightarrow$ (β, α),*p*$\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t*
and *simple-ruleset* (*transform-remove-unknowns-generic* (β, α) *rs*)
and $\forall$ *r* $\in$ *set rs*. $\neg$ *has-disc disc* (*get-match r*) $\Longrightarrow$ $\forall$ *r* $\in$ *set* (*transform-remove-unknowns-generic* (β, α) *rs*). $\neg$ *has-disc disc* (*get-match r*)
and $\forall$ *r* $\in$ *set* (*transform-remove-unknowns-generic* (β, α) *rs*). $\neg$ *has-unknowns* β (*get-match r*)

and $\forall$ *r* $\in$ *set rs*. *normalized-n-primitive disc-sel f* (*get-match r*) $\Longrightarrow$ $\forall$ *r* $\in$ *set* (*transform-remove-unknowns-generic* (β, α) *rs*). *normalized-n-primitive disc-sel f* (*get-match r*)
and $\forall$ *r* $\in$ *set rs*. $\neg$ *has-disc-negated disc neg* (*get-match r*) $\Longrightarrow$ $\forall$ *r* $\in$ *set* (*transform-remove-unknowns-generic* (β, α) *rs*). $\neg$ *has-disc-negated disc neg* (*get-match r*)
proof −
let *?*γ=(β, α)
let *?fw*=λ*rs*. *approximating-bigstep-fun ?*γ *p rs s*

show *simplers1*: *simple-ruleset* (*transform-remove-unknowns-generic ?*γ *rs*)
unfolding *transform-remove-unknowns-generic-def*
using *simplers optimize-matches-a-simple-ruleset* **by** *blast*

show *?*γ,*p*$\vdash$ $\langle$*transform-remove-unknowns-generic ?*γ *rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t* $\longleftrightarrow$ *?*γ,*p*$\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t*
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers1*]]
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers*]]
unfolding *transform-remove-unknowns-generic-def*
using *optimize-matches-a-simplers*[*OF simplers*] *remove-unknowns-generic* **by** *metis*

from *remove-unknowns-generic-not-has-disc* **show**
$\forall$ *r* $\in$ *set rs*. $\neg$ *has-disc disc* (*get-match r*) $\Longrightarrow$ $\forall$ *r* $\in$ *set* (*transform-remove-unknowns-generic ?*γ *rs*). $\neg$ *has-disc disc* (*get-match r*)
unfolding *transform-remove-unknowns-generic-def*
by(*intro optimize-matches-a-preserves*) *blast*

from *remove-unknowns-generic-normalized-n-primitive* **show**
$\forall$ *r* $\in$ *set rs*. *normalized-n-primitive disc-sel f* (*get-match r*) $\Longrightarrow$

$\forall\ r \in set\ (transform\text{-}remove\text{-}unknowns\text{-}generic\ ?\gamma\ rs).\ normalized\text{-}n\text{-}primitive\ disc\text{-}sel\ f\ (get\text{-}match\ r)$
unfolding *transform-remove-unknowns-generic-def*
by(*intro optimize-matches-a-preserves*) *blast*

show $\forall\ r \in set\ (transform\text{-}remove\text{-}unknowns\text{-}generic\ ?\gamma\ rs).\ \neg\ has\text{-}unknowns\ \beta\ (get\text{-}match\ r)$
unfolding *transform-remove-unknowns-generic-def*
apply(*rule optimize-matches-a-preserves*)
apply(*rule remove-unknowns-generic-specification*[*OF - packet-independent-*α *wf*β])
using *simplers* **by**(*simp add*: *simple-ruleset-def*)

from *remove-unknowns-generic-not-has-disc-negated* **show**
$\forall\ r \in set\ rs.\ \neg\ has\text{-}disc\text{-}negated\ disc\ neg\ (get\text{-}match\ r) \Longrightarrow$
$\forall\ r \in set\ (transform\text{-}remove\text{-}unknowns\text{-}generic\ ?\gamma\ rs).\ \neg\ has\text{-}disc\text{-}negated\ disc\ neg\ (get\text{-}match\ r)$
unfolding *transform-remove-unknowns-generic-def*
by(*rule optimize-matches-a-preserves*) *blast*
qed
thm *transform-remove-unknowns-generic*[*OF - - - packet-independent-*β*-unknown-common-matcher*]

corollary *transform-remove-unknowns-upper*: **defines** *upper* $\equiv$ *optimize-matches-a upper-closure-matchexpr*
assumes *simplers*: *simple-ruleset rs*
shows (*common-matcher*, *in-doubt-allow*),*p*$\vdash$ $\langle upper\ rs,\ s\rangle \Rightarrow_\alpha t \longleftrightarrow$ (*common-matcher*, *in-doubt-allow*),*p*$\vdash$ $\langle rs,\ s\rangle \Rightarrow_\alpha t$
and *simple-ruleset* (*upper rs*)
and $\forall\ r \in set\ rs.\ \neg\ has\text{-}disc\ disc\ (get\text{-}match\ r) \Longrightarrow$
$\forall\ r \in set\ (upper\ rs).\ \neg\ has\text{-}disc\ disc\ (get\text{-}match\ r)$
and $\forall\ r \in set\ (upper\ rs).\ \neg\ has\text{-}disc\ is\text{-}Extra\ (get\text{-}match\ r)$
and $\forall\ r \in set\ rs.\ normalized\text{-}n\text{-}primitive\ disc\text{-}sel\ f\ (get\text{-}match\ r) \Longrightarrow$
$\forall\ r \in set\ (upper\ rs).\ normalized\text{-}n\text{-}primitive\ disc\text{-}sel\ f\ (get\text{-}match\ r)$
and $\forall\ r \in set\ rs.\ \neg\ has\text{-}disc\text{-}negated\ disc\ neg\ (get\text{-}match\ r) \Longrightarrow$
$\forall\ r \in set\ (upper\ rs).\ \neg\ has\text{-}disc\text{-}negated\ disc\ neg\ (get\text{-}match\ r)$
proof −
from *simplers* **have** *upper*: *upper rs* = *transform-remove-unknowns-generic* (*common-matcher*, *in-doubt-allow*) *rs*
apply(*simp add*: *transform-remove-unknowns-generic-def upper-def*)
apply(*erule optimize-matches-a-simple-ruleset-eq*)
by (*simp add*: *upper-closure-matchexpr-generic*)

note $*$ = *transform-remove-unknowns-generic*[*OF simplers wf-in-doubt-allow packet-independent-unknown-match-tacs*(*1*) *packet-independent-*β*-unknown-com simplified upper-closure-matchexpr-generic*]

from $*(1)$[**where** $p = p$]
show (*common-matcher*, *in-doubt-allow*),*p*$\vdash$ $\langle upper\ rs,\ s\rangle \Rightarrow_\alpha t \longleftrightarrow$ (*common-matcher*,

in-doubt-allow),*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t*
by(*subst upper*)
from ∗(*2*) **show** *simple-ruleset* (*upper rs*) **by**(*subst upper*)
from ∗(*3*) **show** ∀ *r* ∈ *set rs*. ¬ *has-disc disc* (*get-match r*) ⟹
∀ *r* ∈ *set* (*upper rs*). ¬ *has-disc disc* (*get-match r*)
by(*subst upper*) *fast*
from ∗(*4*) **show** ∀ *r* ∈ *set* (*upper rs*). ¬ *has-disc is-Extra* (*get-match r*)
apply(*subst upper*)
using *has-unknowns-common-matcher* **by** *auto*
from ∗(*5*) **show** ∀ *r* ∈ *set rs. normalized-n-primitive disc-sel f* (*get-match r*)
⟹
∀ *r* ∈ *set* (*upper rs*). *normalized-n-primitive disc-sel f* (*get-match r*)
apply(*subst upper*)
using *packet-independent-unknown-match-tacs*(*1*) *simplers*
transform-remove-unknowns-generic(*5*)[*OF - - - packet-independent-β-unknown-common-matcher*]
wf-in-doubt-allow
by *blast*
from ∗(*6*) **show** ∀ *r* ∈ *set rs*. ¬ *has-disc-negated disc neg* (*get-match r*) ⟹
∀ *r* ∈ *set* (*upper rs*). ¬ *has-disc-negated disc neg* (*get-match r*)
by(*subst upper*) *fast*
qed

corollary *transform-remove-unknowns-lower*: **defines** *lower* ≡ *optimize-matches-a*
lower-closure-matchexpr
assumes *simplers*: *simple-ruleset rs*
shows (*common-matcher*, *in-doubt-deny*),*p*⊢ ⟨*lower rs*, *s*⟩ ⇒$_\alpha$ *t* ⟷ (*common-matcher*,
in-doubt-deny),*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t*
and *simple-ruleset* (*lower rs*)
and ∀ *r* ∈ *set rs*. ¬ *has-disc disc* (*get-match r*) ⟹
∀ *r* ∈ *set* (*lower rs*). ¬ *has-disc disc* (*get-match r*)
and ∀ *r* ∈ *set* (*lower rs*). ¬ *has-disc is-Extra* (*get-match r*)
and ∀ *r* ∈ *set rs. normalized-n-primitive disc-sel f* (*get-match r*) ⟹
∀ *r* ∈ *set* (*lower rs*). *normalized-n-primitive disc-sel f* (*get-match r*)
and ∀ *r* ∈ *set rs*. ¬ *has-disc-negated disc neg* (*get-match r*) ⟹
∀ *r* ∈ *set* (*lower rs*). ¬ *has-disc-negated disc neg* (*get-match r*)
proof −
from *simplers* **have** *lower*: *lower rs* = *transform-remove-unknowns-generic* (*common-matcher*,
in-doubt-deny) *rs*
apply(*simp add*: *transform-remove-unknowns-generic-def lower-def*)
apply(*erule optimize-matches-a-simple-ruleset-eq*)
by (*simp add*: *lower-closure-matchexpr-generic*)

note ∗ = *transform-remove-unknowns-generic*[*OF*
simplers wf-in-doubt-deny packet-independent-unknown-match-tacs(*2*) *packet-independent-β-unknown-comm*
simplified lower-closure-matchexpr-generic]

from ∗(*1*)[**where** *p* = *p*]

show (*common-matcher*, *in-doubt-deny*),*p*⊢ ⟨*lower rs*, *s*⟩ ⇒$_\alpha$ *t* ⟷ (*common-matcher*, *in-doubt-deny*),*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t*
by(*subst lower*)
from ∗(*2*) **show** *simple-ruleset* (*lower rs*) **by**(*subst lower*)
from ∗(*3*) **show** ∀ *r* ∈ *set rs*. ¬ *has-disc disc* (*get-match r*) ⟹
∀ *r* ∈ *set* (*lower rs*). ¬ *has-disc disc* (*get-match r*)
by(*subst lower*) *fast*
from ∗(*4*) **show** ∀ *r* ∈ *set* (*lower rs*). ¬ *has-disc is-Extra* (*get-match r*)
apply(*subst lower*)
using *has-unknowns-common-matcher* **by** *auto*
from ∗(*5*) **show** ∀ *r* ∈ *set rs*. *normalized-n-primitive disc-sel f* (*get-match r*) ⟹
∀ *r* ∈ *set* (*lower rs*). *normalized-n-primitive disc-sel f* (*get-match r*)
apply(*subst lower*)
using *packet-independent-unknown-match-tacs*(*1*) *simplers*
transform-remove-unknowns-generic(*5*)[*OF - - - packet-independent-β-unknown-common-matcher*] *wf-in-doubt-deny*
by *blast*
from ∗(*6*) **show** ∀ *r* ∈ *set rs*. ¬ *has-disc-negated disc neg* (*get-match r*) ⟹
∀ *r* ∈ *set* (*lower rs*). ¬ *has-disc-negated disc neg* (*get-match r*)
by(*subst lower*) *fast*
qed

39.4 Normalizing and Transforming Primitives

Rewrite the primitives IPs and Ports such that can be used by the simple firewall.

definition *transform-normalize-primitives* :: *'i*::*len common-primitive rule list* ⇒ *'i common-primitive rule list* **where**
transform-normalize-primitives =
optimize-matches-option compress-normalize-besteffort ∘ — normalizes protocols, needs to go last
normalize-rules normalize-dst-ips ∘
normalize-rules normalize-src-ips ∘
normalize-rules normalize-dst-ports ∘ — may introduce new matches on protocols
normalize-rules normalize-src-ports ∘ — may introduce new matches in protocols
normalize-rules rewrite-MultiportPorts — introduces *Src-Ports* and *Dst-Ports* matches

thm *normalize-primitive-extract-preserves-unrelated-normalized-n-primitive*
lemma *normalize-rules-preserves-unrelated-normalized-n-primitive*:
assumes ∀ *r* ∈ *set rs*. *normalized-nnf-match* (*get-match r*) ∧ *normalized-n-primitive* (*disc2*, *sel2*) *P* (*get-match r*)
and *wf-disc-sel* (*disc1*, *sel1*) *C*
and ∀ *a*. ¬ *disc2* (*C a*)
shows ∀ *r* ∈ *set* (*normalize-rules* (*normalize-primitive-extract* (*disc1*, *sel1*) *C*

f) *rs*).
normalized-nnf-match (*get-match r*) ∧ *normalized-n-primitive* (*disc2*, *sel2*) *P* (*get-match r*)
thm *normalize-rules-preserves*[**where** *P*=λ*m*. *normalized-nnf-match m* ∧ *normalized-n-primitive* (*disc2*, *sel2*) *P m*
and *f*=*normalize-primitive-extract* (*disc1*, *sel1*) *C f*]
apply(*rule normalize-rules-preserves*[**where** *P*=λ*m*. *normalized-nnf-match m* ∧ *normalized-n-primitive* (*disc2*, *sel2*) *P m*
and *f*=*normalize-primitive-extract* (*disc1*, *sel1*) *C f*])
using *assms*(*1*) **apply**(*simp*)
apply(*safe*)
using *normalize-primitive-extract-preserves-nnf-normalized*[*OF* - *assms*(*2*)]
apply *fast*
using *normalize-primitive-extract-preserves-unrelated-normalized-n-primitive*[*OF* - - *assms*(*2*) *assms*(*3*)] **by** *blast*

lemma *normalize-rules-normalized-n-primitive*:
assumes ∀ *r* ∈ *set rs*. *normalized-nnf-match* (*get-match r*)
and ∀ *m*. *normalized-nnf-match m* ⟶
(∀ *m'* ∈ *set* (*normalize-primitive-extract* (*disc*, *sel*) *C f m*). *normalized-n-primitive* (*disc*, *sel*) *P m'*)
shows ∀ *r* ∈ *set* (*normalize-rules* (*normalize-primitive-extract* (*disc*, *sel*) *C f*) *rs*).
normalized-n-primitive (*disc*, *sel*) *P* (*get-match r*)
apply(*rule normalize-rules-property*[**where** *P*=*normalized-nnf-match* **and** *f*=*normalize-primitive-extract* (*disc*, *sel*) *C f*])
using *assms*(*1*) **apply** *simp*
using *assms*(*2*) **by** *simp*

lemma *optimize-matches-option-compress-normalize-besteffort-preserves-unrelated-normalized-n-primitive*:
assumes ∀ *r* ∈ *set rs*. *normalized-nnf-match* (*get-match r*) ∧ *normalized-n-primitive* (*disc2*, *sel2*) *P* (*get-match r*)
and ∀ *a*. ¬ *disc2* (*IIface a*) **and** ∀ *a*. ¬ *disc2* (*OIface a*) **and** ∀ *a*. ¬ *disc2* (*Prot a*)
shows ∀ *r* ∈ *set* (*optimize-matches-option compress-normalize-besteffort rs*).
normalized-nnf-match (*get-match r*) ∧ *normalized-n-primitive* (*disc2*, *sel2*) *P* (*get-match r*)
thm *optimize-matches-option-preserves*
apply(*rule optimize-matches-option-preserves*[**where** *P*=λ*m*. *normalized-nnf-match m* ∧ *normalized-n-primitive* (*disc2*, *sel2*) *P m*
and *f*=*compress-normalize-besteffort*])
apply(*rule compress-normalize-besteffort-preserves-normalized-n-primitive*)
apply(*simp-all add*: *assms*)
done

theorem *transform-normalize-primitives*:
— all discriminators which will not be normalized remain unchanged

defines *unchanged disc* ≡ (∀ *a*. ¬ *disc* (*Src-Ports a*)) ∧ (∀ *a*. ¬ *disc* (*Dst-Ports a*)) ∧
(∀ *a*. ¬ *disc* (*Src a*)) ∧ (∀ *a*. ¬ *disc* (*Dst a*))
— also holds for these discriminators, but not for *Prot*, which might be changed
and *changeddisc disc* ≡ ((∀ *a*. ¬ *disc* (*IIface a*)) ∨ *disc* = *is-Iiface*) ∧
((∀ *a*. ¬ *disc* (*OIface a*)) ∨ *disc* = *is-Oiface*)

assumes *simplers*: *simple-ruleset* (*rs* :: *'i*::*len common-primitive rule list*)
and *wfα*: *wf-unknown-match-tac α*
and *normalized*: ∀ *r* ∈ *set rs*. *normalized-nnf-match* (*get-match r*)
shows (*common-matcher*, *α*),*p*⊢ ⟨*transform-normalize-primitives rs*, *s*⟩ ⇒*α t* ⟷ (*common-matcher*, *α*),*p*⊢ ⟨*rs*, *s*⟩ ⇒*α t*
and *simple-ruleset* (*transform-normalize-primitives rs*)
and *unchanged disc1* ⟹ *changeddisc disc1* ⟹ ∀ *a*. ¬ *disc1* (*Prot a*) ⟹
∀ *r* ∈ *set rs*. ¬ *has-disc disc1* (*get-match r*) ⟹
∀ *r* ∈ *set* (*transform-normalize-primitives rs*). ¬ *has-disc disc1* (*get-match r*)
and ∀ *r* ∈ *set* (*transform-normalize-primitives rs*). *normalized-nnf-match* (*get-match r*)
and ∀ *r* ∈ *set* (*transform-normalize-primitives rs*).
normalized-src-ports (*get-match r*) ∧ *normalized-dst-ports* (*get-match r*) ∧
normalized-src-ips (*get-match r*) ∧ *normalized-dst-ips* (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*)
and *unchanged disc2* ⟹ (∀ *a*. ¬ *disc2* (*IIface a*)) ⟹ (∀ *a*. ¬ *disc2* (*OIface a*)) ⟹ (∀ *a*. ¬ *disc2* (*Prot a*)) ⟹
∀ *r* ∈ *set rs*. *normalized-n-primitive* (*disc2*, *sel2*) *f* (*get-match r*) ⟹
∀ *r* ∈ *set* (*transform-normalize-primitives rs*). *normalized-n-primitive* (*disc2*, *sel2*) *f* (*get-match r*)
— For disc3, we do not allow ports and ips, because these are changed. Here is the complicated part: (It is only complicated if, basically disc3 is *is-Prot*) In addition, either it must not be protocol or (complicated case) there must be no negated port matches in the ruleset. Note that negated *Src-Ports* or *Dst-Ports* can also be introduced by rewriting *MultiportPorts*
and *unchanged disc3* ⟹ *changeddisc disc3* ⟹
(∀ *a*. ¬ *disc3* (*Prot a*)) ∨
(*disc3* = *is-Prot* ∧ (∀ *r* ∈ *set rs*.
¬ *has-disc-negated is-Src-Ports False* (*get-match r*) ∧
¬ *has-disc-negated is-Dst-Ports False* (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*))) ⟹
∀ *r* ∈ *set rs*. ¬ *has-disc-negated disc3 False* (*get-match r*) ⟹
∀ *r* ∈ *set* (*transform-normalize-primitives rs*). ¬ *has-disc-negated disc3 False* (*get-match r*)
proof −
let *?γ*=(*common-matcher*, *α*)
let *?fw*=λ*rs*. *approximating-bigstep-fun ?γ p rs s*

show *simplers-t*: *simple-ruleset* (*transform-normalize-primitives rs*)
unfolding *transform-normalize-primitives-def*
by(*simp add*: *simple-ruleset-normalize-rules simplers optimize-matches-option-simple-ruleset*)

let *?rs0=normalize-rules rewrite-MultiportPorts rs*
let *?rs1=normalize-rules normalize-src-ports ?rs0*
let *?rs2=normalize-rules normalize-dst-ports ?rs1*
let *?rs3=normalize-rules normalize-src-ips ?rs2*
let *?rs4=normalize-rules normalize-dst-ips ?rs3*
let *?rs5=optimize-matches-option compress-normalize-besteffort ?rs4*

have *normalized-rs0*: $\forall r \in$ *set ?rs0*. *normalized-nnf-match* (*get-match r*)
apply(*intro normalize-rules-preserves*[*OF normalized*])
apply(*simp add*: *rewrite-MultiportPorts-def*)
using *normalized-nnf-match-normalize-match* **by** *blast*
from *normalize-src-ports-nnf* **have** *normalized-rs1*: $\forall r \in$ *set ?rs1*. *normalized-nnf-match* (*get-match r*)
apply(*intro normalize-rules-preserves*[*OF normalized-rs0*])
using *normalize-dst-ports-nnf* **by** *blast*
from *normalize-dst-ports-nnf* **have** *normalized-rs2*: $\forall r \in$ *set ?rs2*. *normalized-nnf-match* (*get-match r*)
apply(*intro normalize-rules-preserves*[*OF normalized-rs1*])
by *blast*
from *normalize-rules-primitive-extract-preserves-nnf-normalized*[*OF this wf-disc-sel-common-primitive*(*3*)]
normalize-src-ips-def
have *normalized-rs3*: $\forall r \in$ *set ?rs3*. *normalized-nnf-match* (*get-match r*) **by** *metis*
from *normalize-rules-primitive-extract-preserves-nnf-normalized*[*OF this wf-disc-sel-common-primitive*(*4*)]
normalize-dst-ips-def
have *normalized-rs4*: $\forall r \in$ *set ?rs4*. *normalized-nnf-match* (*get-match r*) **by** *metis*
have *normalized-rs5*: $\forall r \in$ *set ?rs5*. *normalized-nnf-match* (*get-match r*)
apply(*intro optimize-matches-option-preserves*)
apply(*erule compress-normalize-besteffort-nnf*[*rotated*])
by(*simp add*: *normalized-rs4*)
thus $\forall\ r \in$ *set* (*transform-normalize-primitives rs*). *normalized-nnf-match* (*get-match r*)
unfolding *transform-normalize-primitives-def* **by** *simp*

have *local-simp*: $\bigwedge$*rs1 rs2*. *approximating-bigstep-fun ?γ p rs1 s* = *approximating-bigstep-fun ?γ p rs2 s* $\Longrightarrow$
(*approximating-bigstep-fun ?γ p rs1 s* = *t*) = (*approximating-bigstep-fun ?γ p rs2 s* = *t*) **by** *simp*

have *opt-compress-rule*:
approximating-bigstep-fun (*common-matcher*, α) *p* (*optimize-matches-option compress-normalize-besteffort rs1*) *s* =
approximating-bigstep-fun (*common-matcher*, α) *p rs2 s*
if *rs1-n*: $\forall r \in$ *set rs1*. *normalized-nnf-match* (*get-match r*)
and *rs1rs2*: *approximating-bigstep-fun* (*common-matcher*, α) *p rs1 s* =
approximating-bigstep-fun (*common-matcher*, α) *p rs2 s* **for** *rs1 rs2*

apply(*subst optimize-matches-option-generic*[**where** *P*=λ *m a. normalized-nnf-match m*])
apply(*simp-all add*: *normalized*
compress-normalize-besteffort-Some[*OF primitive-matcher-generic-common-matcher*]
compress-normalize-besteffort-None[*OF primitive-matcher-generic-common-matcher*]
rs1-n)
using *rs1rs2* **by** *simp*

show *?γ,p*⊢ ⟨*transform-normalize-primitives rs*, *s*⟩ ⇒$_\alpha$ *t* ⟷ *?γ,p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t*
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers-t*]]
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers*]]
unfolding *transform-normalize-primitives-def*
apply(*simp*)
apply(*subst local-simp*, *simp-all*)
apply(*rule opt-compress-rule*[*OF normalized-rs4*])
apply(*subst normalize-rules-match-list-semantics-3*[*of normalized-nnf-match*])
using *normalize-dst-ips*[**where** *p* = *p*] **apply**(*simp*; *fail*)
using *simplers simple-ruleset-normalize-rules* **apply** *blast*
using *normalized-rs3* **apply**(*simp*; *fail*)
apply(*subst normalize-rules-match-list-semantics-3*[*of normalized-nnf-match*])
using *normalize-src-ips*[**where** *p* = *p*] **apply**(*simp*; *fail*)
using *simplers simple-ruleset-normalize-rules* **apply** *blast*
using *normalized-rs2* **apply**(*simp*; *fail*)
apply(*subst normalize-rules-match-list-semantics-3*[*of normalized-nnf-match*])
using *normalize-dst-ports*[*OF primitive-matcher-generic-common-matcher*,**where** *p* = *p*] **apply**(*simp*; *fail*)
using *simplers simple-ruleset-normalize-rules* **apply** *blast*
using *normalized-rs1* **apply**(*simp*; *fail*)
apply(*subst normalize-rules-match-list-semantics-3*[*of normalized-nnf-match*])
using *normalize-src-ports*[*OF primitive-matcher-generic-common-matcher*, **where** *p* = *p*] **apply**(*simp*; *fail*)
using *simplers simple-ruleset-normalize-rules* **apply** *blast*
using *normalized-rs0* **apply**(*simp*; *fail*)
apply(*subst normalize-rules-match-list-semantics-3*[*of normalized-nnf-match*])
using *rewrite-MultiportPorts*[*OF primitive-matcher-generic-common-matcher*, **where** *p* = *p*] **apply**(*simp*; *fail*)
using *simplers* **apply** *blast*
using *normalized* **apply**(*simp*; *fail*)
..

from *rewrite-MultiportPorts-removes-MultiportsPorts*
normalize-rules-property[*OF normalized*, **where** *f*=*rewrite-MultiportPorts* **and** *Q*=λ*m*. ¬ *has-disc is-MultiportPorts m*]
have *rewrite-MultiportPorts-normalizes-Multiports*:
∀ *r* ∈ *set ?rs0*. ¬ *has-disc is-MultiportPorts* (*get-match r*)

by *blast*
from *normalize-src-ports-normalized-n-primitive*
have *normalized-src-ports*: $\forall r \in$ *set ?rs1*. *normalized-src-ports* (*get-match r*)
apply(*intro normalize-rules-property*[*OF normalized-rs0*, **where** *f*=*normalize-src-ports* **and** *Q*=*normalized-src-ports*])
by *blast*
from *normalize-dst-ports-normalized-n-primitive*
normalize-rules-property[*OF normalized-rs1*, **where** *f*=*normalize-dst-ports* **and** *Q*=*normalized-dst-ports*]
have *normalized-dst-ports*: $\forall r \in$ *set ?rs2*. *normalized-dst-ports* (*get-match r*) **by** *fast*
from *normalize-src-ips-normalized-n-primitive*
normalize-rules-property[*OF normalized-rs2*, **where** *f*=*normalize-src-ips* **and** *Q*=*normalized-src-ips*]
have *normalized-src-ips*: $\forall r \in$ *set ?rs3*. *normalized-src-ips* (*get-match r*) **by** *fast*
from *normalize-dst-ips-normalized-n-primitive*
normalize-rules-property[*OF normalized-rs3*, **where** *f*=*normalize-dst-ips* **and** *Q*=*normalized-dst-ips*]
normalized-rs4
have *normalized-dst-ips-rs4*: $\forall r \in$ *set ?rs4*. *normalized-nnf-match* (*get-match r*) $\wedge$ *normalized-dst-ips* (*get-match r*) **by** *fast*
with *optimize-matches-option-compress-normalize-besteffort-preserves-unrelated-normalized-n-primitive*[*of - is-Dst dst-sel normalized-cidr-ip*, *folded normalized-dst-ips-def2*]
have *normalized-dst-rs5*: $\forall r \in$ *set ?rs5*. *normalized-dst-ips* (*get-match r*) **by** *fastforce*

have *normalize-dst-ports-preserves-normalized-src-ports*:
$m' \in$ *set* (*normalize-dst-ports m*) $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *normalized-src-ports m* $\Longrightarrow$ *normalized-src-ports m'* **for** *m m'* :: *'i common-primitive match-expr*
unfolding *normalized-src-ports-def2*
apply(*rule normalize-ports-generic-preserves-normalized-n-primitive*[*OF - wf-disc-sel-common-primitive*(*2*)])
apply(*simp-all*)
by (*simp add*: *normalize-dst-ports-def normalize-ports-generic-def normalize-positive-dst-ports-def rewrite-negated-dst-ports-def*)

from *normalize-rules-preserves-unrelated-normalized-n-primitive*[*of - is-MultiportPorts multiportports-sel* λ*-. False*]
have *preserve-normalized-multiport-ports*:
$\forall r \in$ *set rs*. *normalized-nnf-match* (*get-match r*) $\Longrightarrow$
$\forall r \in$ *set rs*. $\neg$ *has-disc is-MultiportPorts* (*get-match r*) $\Longrightarrow$
wf-disc-sel (*disc*, *sel*) *C* $\Longrightarrow$
$\forall a$. $\neg$ *is-MultiportPorts* (*C a*) $\Longrightarrow$
$\forall r \in$ *set* (*normalize-rules* (*normalize-primitive-extract* (*disc*, *sel*) *C f*) *rs*).
$\neg$ *has-disc is-MultiportPorts* (*get-match r*)

for $f :: 'c$ *negation-type list* $\Rightarrow 'c$ *list* **and** *rs disc sel*
and $C :: 'c \Rightarrow 'i$::*len common-primitive*
using *normalized-n-primitive-false-eq-notdisc*
by *blast*

have *normalized-multiportports-rs1*: $\forall r \in$ *set ?rs1*. $\neg$ *has-disc is-MultiportPorts* (*get-match r*)
apply(*rule normalize-rules-property*[**where** *P*=λm. *normalized-nnf-match m* $\wedge \neg$ *has-disc is-MultiportPorts m*])
using *normalized-rs0 rewrite-MultiportPorts-normalizes-Multiports* **apply** *blast*
apply(*intro allI impI ballI*)
apply(*rule normalize-src-ports-preserves-normalized-not-has-disc*)
by(*simp-all*)
have *normalized-multiportports-rs2*: $\forall r \in$ *set ?rs2*. $\neg$ *has-disc is-MultiportPorts* (*get-match r*)
apply(*rule normalize-rules-property*[**where** *P*=λm. *normalized-nnf-match m* $\wedge \neg$ *has-disc is-MultiportPorts m*])
using *normalized-rs1 normalized-multiportports-rs1* **apply** *blast*
apply(*intro allI impI ballI*)
apply(*rule normalize-dst-ports-preserves-normalized-not-has-disc*)
by(*simp-all*)
from *preserve-normalized-multiport-ports*[*OF normalized-rs2 normalized-multiportports-rs2 wf-disc-sel-common-primitive*(*3*),
where *f2*=*ipt-iprange-compress*, *folded normalize-src-ips-def*]
have *normalized-multiportports-rs3*: $\forall r \in$ *set ?rs3*. $\neg$ *has-disc is-MultiportPorts* (*get-match r*) **by** *simp*
from *preserve-normalized-multiport-ports*[*OF normalized-rs3 normalized-multiportports-rs3 wf-disc-sel-common-primitive*(*4*),
where *f2*=*ipt-iprange-compress*, *folded normalize-dst-ips-def*]
normalized-rs4
have *normalized-multiportports-rs4*: $\forall r \in$ *set ?rs4*. *normalized-nnf-match* (*get-match r*) $\wedge \neg$ *has-disc is-MultiportPorts* (*get-match r*) **by** *simp*
with *optimize-matches-option-compress-normalize-besteffort-preserves-unrelated-normalized-n-primitive*[
of - is-MultiportPorts multiportports-sel λ*-. False*
, *simplified*]
have *normalized-multiportports-rs5*: $\forall r \in$ *set ?rs5*. $\neg$ *has-disc is-MultiportPorts* (*get-match r*)
using *normalized-n-primitive-false-eq-notdisc* **by** *fastforce*

from *normalize-rules-preserves-unrelated-normalized-n-primitive*[*of - is-Src-Ports src-ports-sel* (λ*ps. case ps of L4Ports - pts* $\Rightarrow$ *length pts* $\leq$ *1*),
folded normalized-src-ports-def2]
have *preserve-normalized-src-ports*:
$\forall r \in$ *set rs. normalized-nnf-match* (*get-match r*) $\Longrightarrow$
$\forall r \in$ *set rs. normalized-src-ports* (*get-match r*) $\Longrightarrow$
wf-disc-sel (*disc*, *sel*) *C* $\Longrightarrow$
$\forall a. \neg$ *is-Src-Ports* (*C a*) $\Longrightarrow$
$\forall r \in$ *set* (*normalize-rules* (*normalize-primitive-extract* (*disc*, *sel*) *C f*) *rs*).

normalized-src-ports (*get-match r*)
for *f* :: *'c negation-type list* ⇒ *'c list* **and** *rs disc sel* **and** *C* :: *'c* ⇒ *'i*::*len common-primitive*
by *blast*
have *normalized-src-ports-rs2*: ∀ *r* ∈ *set ?rs2*. *normalized-src-ports* (*get-match r*)
apply(*rule normalize-rules-property*[**where** *P*=λ*m*. *normalized-nnf-match m* ∧ *normalized-src-ports m*])
using *normalized-rs1 normalized-src-ports* **apply** *blast*
apply(*clarify*)
using *normalize-dst-ports-preserves-normalized-src-ports* **by** *blast*
from *preserve-normalized-src-ports*[*OF normalized-rs2 normalized-src-ports-rs2 wf-disc-sel-common-primitive*(*3*),
where *f3*=*ipt-iprange-compress*, *folded normalize-src-ips-def*]
have *normalized-src-ports-rs3*: ∀ *r* ∈ *set ?rs3*. *normalized-src-ports* (*get-match r*) **by** *simp*
from *preserve-normalized-src-ports*[*OF normalized-rs3 normalized-src-ports-rs3 wf-disc-sel-common-primitive*(*4*),
where *f3*=*ipt-iprange-compress*, *folded normalize-dst-ips-def*]
normalized-rs4
have *normalized-src-ports-rs4*: ∀ *r* ∈ *set ?rs4*. *normalized-nnf-match* (*get-match r*) ∧ *normalized-src-ports* (*get-match r*) **by** *simp*
with *optimize-matches-option-compress-normalize-besteffort-preserves-unrelated-normalized-n-primitive*[
of - is-Src-Ports src-ports-sel (λ*ps*. *case ps of L4Ports - pts* ⇒ *length pts* ≤ *1*)
, *folded normalized-src-ports-def2*]
have *normalized-src-ports-rs5*: ∀ *r* ∈ *set ?rs5*. *normalized-src-ports* (*get-match r*) **by** *fastforce*

from *normalize-rules-preserves-unrelated-normalized-n-primitive*[*of - is-Dst-Ports dst-ports-sel* (λ*ps*. *case ps of L4Ports - pts* ⇒ *length pts* ≤ *1*),
folded normalized-dst-ports-def2]
have *preserve-normalized-dst-ports*: ⋀*rs disc sel C f*.
∀ *r*∈*set rs*. *normalized-nnf-match* (*get-match r*) ⟹
∀ *r*∈*set rs*. *normalized-dst-ports* (*get-match r*) ⟹
wf-disc-sel (*disc*, *sel*) *C* ⟹
∀ *a*. ¬ *is-Dst-Ports* (*C a*) ⟹
∀ *r*∈ *set* (*normalize-rules* (*normalize-primitive-extract* (*disc*, *sel*) *C f*) *rs*). *normalized-dst-ports* (*get-match r*)
by *blast*
from *preserve-normalized-dst-ports*[*OF normalized-rs2 normalized-dst-ports wf-disc-sel-common-primitive*(*3*
where *f3*=*ipt-iprange-compress*, *folded normalize-src-ips-def*]
have *normalized-dst-ports-rs3*: ∀ *r* ∈ *set ?rs3*. *normalized-dst-ports* (*get-match r*) **by** *force*
from *preserve-normalized-dst-ports*[*OF normalized-rs3 normalized-dst-ports-rs3 wf-disc-sel-common-primitive*(*4*),
where *f3*=*ipt-iprange-compress*, *folded normalize-dst-ips-def*]
normalized-rs4
have *normalized-dst-ports-rs4*: ∀ *r* ∈ *set ?rs4*. *normalized-nnf-match* (*get-match*

$r) \wedge$ *normalized-dst-ports* (*get-match r*) **by** *force*
with *optimize-matches-option-compress-normalize-besteffort-preserves-unrelated-normalized-n-primitive*[
of - is-Dst-Ports dst-ports-sel (λ*ps. case ps of L4Ports - pts* $\Rightarrow$ *length pts* $\leq$ *1*)
, *folded normalized-dst-ports-def2*]
have *normalized-dst-ports-rs5*: $\forall$ *r* $\in$ *set ?rs5*. *normalized-dst-ports* (*get-match r*) **by** *fastforce*

from *normalize-rules-preserves-unrelated-normalized-n-primitive*[*of ?rs3 is-Src src-sel normalized-cidr-ip*,
OF - wf-disc-sel-common-primitive(*4*),
where *f*=*ipt-iprange-compress*, *folded normalize-dst-ips-def normalized-src-ips-def2*]
normalized-rs3 normalized-src-ips
have *normalized-src-rs4*: $\forall$ *r* $\in$ *set ?rs4*. *normalized-nnf-match* (*get-match r*) $\wedge$ *normalized-src-ips* (*get-match r*) **by** *force*
with *optimize-matches-option-compress-normalize-besteffort-preserves-unrelated-normalized-n-primitive*[
of - is-Src src-sel normalized-cidr-ip
, *folded normalized-src-ips-def2*]
have *normalized-src-rs5*: $\forall$ *r* $\in$ *set ?rs5*. *normalized-src-ips* (*get-match r*)
by *fastforce*

from *normalized-multiportports-rs5 normalized-src-ports-rs5 normalized-dst-ports-rs5 normalized-src-rs5 normalized-dst-rs5*
show $\forall$ *r* $\in$ *set* (*transform-normalize-primitives rs*).
normalized-src-ports (*get-match r*) $\wedge$ *normalized-dst-ports* (*get-match r*) $\wedge$
normalized-src-ips (*get-match r*) $\wedge$ *normalized-dst-ips* (*get-match r*) $\wedge$
$\neg$ *has-disc is-MultiportPorts* (*get-match r*)
unfolding *transform-normalize-primitives-def* **by** *simp*

show *unchanged disc2* $\Longrightarrow$ ($\forall$ *a*. $\neg$ *disc2* (*IIface a*)) $\Longrightarrow$ ($\forall$ *a*. $\neg$ *disc2* (*OIface a*)) $\Longrightarrow$ ($\forall$ *a*. $\neg$ *disc2* (*Prot a*)) $\Longrightarrow$
$\forall$ *r* $\in$ *set rs*. *normalized-n-primitive* (*disc2*, *sel2*) *f* (*get-match r*) $\Longrightarrow$
$\forall$ *r* $\in$ *set* (*transform-normalize-primitives rs*). *normalized-n-primitive* (*disc2*, *sel2*) *f* (*get-match r*)
unfolding *unchanged-def*
proof(*elim conjE*)
assume $\forall$ *r*$\in$ *set rs*. *normalized-n-primitive* (*disc2*, *sel2*) *f* (*get-match r*)
with *normalized* **have** *a'*:
$\forall$ *r*$\in$ *set rs*. *normalized-nnf-match* (*get-match r*) $\wedge$ *normalized-n-primitive* (*disc2*, *sel2*) *f* (*get-match r*) **by** *blast*

assume *a-Src-Ports*: $\forall$ *a*. $\neg$ *disc2* (*Src-Ports a*)
assume *a-Dst-Ports*: $\forall$ *a*. $\neg$ *disc2* (*Dst-Ports a*)
assume *a-Src*: $\forall$ *a*. $\neg$ *disc2* (*Src a*)
assume *a-Dst*: $\forall$ *a*. $\neg$ *disc2* (*Dst a*)
assume *a-IIface*: ($\forall$ *a*. $\neg$ *disc2* (*IIface a*))
assume *a-OIface*: ($\forall$ *a*. $\neg$ *disc2* (*OIface a*))
assume *a-Prot*: ($\forall$ *a*. $\neg$ *disc2* (*Prot a*))

have *normalized-n-primitive-rs0*:
∀ *r*∈*set ?rs0*. *normalized-n-primitive* (*disc2*, *sel2*) *f* (*get-match r*)
apply(*intro normalize-rules-property*[**where** *P*=λ*m*. *normalized-nnf-match m* ∧ *normalized-n-primitive* (*disc2*, *sel2*) *f m*])
using *a*′ **apply** *blast*
using *rewrite-MultiportPorts-preserves-normalized-n-primitive*[*OF - a-Src-Ports a-Dst-Ports*] **by** *blast*
have *normalized-n-primitive-rs1*:
∀ *r*∈*set ?rs1*. *normalized-n-primitive* (*disc2*, *sel2*) *f* (*get-match r*)
apply(*rule normalize-rules-property*[**where** *P*=λ*m*. *normalized-nnf-match m* ∧ *normalized-n-primitive* (*disc2*, *sel2*) *f m*])
using *normalized-n-primitive-rs0 normalized-rs0* **apply** *blast*
using *normalize-src-ports-preserves-normalized-n-primitive*[*OF - a-Src-Ports*] *a-Prot* **by** *blast*
have ∀ *r*∈*set ?rs2*. *normalized-n-primitive* (*disc2*, *sel2*) *f* (*get-match r*)
apply(*rule normalize-rules-property*[**where** *P*=λ*m*. *normalized-nnf-match m* ∧ *normalized-n-primitive* (*disc2*, *sel2*) *f m*])
using *normalized-n-primitive-rs1 normalized-rs1* **apply** *blast*
using *normalize-dst-ports-preserves-normalized-n-primitive*[*OF - a-Dst-Ports*] *a-Prot* **by** *blast*
with *normalized-rs2 normalize-rules-preserves-unrelated-normalized-n-primitive*[*OF - wf-disc-sel-common-primitive*(*3*) *a-Src*,
of ?rs2 sel2 f ipt-iprange-compress,
folded normalize-src-ips-def]
have ∀ *r*∈*set ?rs3*. *normalized-n-primitive* (*disc2*, *sel2*) *f* (*get-match r*) **by** *blast*
with *normalized-rs3 normalize-rules-preserves-unrelated-normalized-n-primitive*[*OF - wf-disc-sel-common-primitive*(*4*) *a-Dst*,
of ?rs3 sel2 f ipt-iprange-compress,
folded normalize-dst-ips-def]
have ∀ *r*∈*set ?rs4*. *normalized-nnf-match* (*get-match r*) ∧ *normalized-n-primitive* (*disc2*, *sel2*) *f* (*get-match r*) **by** *blast*
hence ∀ *r*∈*set ?rs5*. *normalized-nnf-match* (*get-match r*) ∧ *normalized-n-primitive* (*disc2*, *sel2*) *f* (*get-match r*)
apply(*intro optimize-matches-option-compress-normalize-besteffort-preserves-unrelated-normalized-n-prim*
using *a-IIface a-OIface a-Prot* **by** *simp-all*
thus *?thesis*
unfolding *transform-normalize-primitives-def* **by** *simp*
qed

— Pushing through properties through the ip normalizer
{ fix *m* **and** *m*′ **and** *disc*::(′*i*::*len common-primitive* ⇒ *bool*)
and *sel*::(′*i*::*len common-primitive* ⇒ ′*x*) **and** *C*′:: (′*x* ⇒ ′*i*::*len common-primitive*)
and *f*′::(′*x negation-type list* ⇒ ′*x list*)
assume *am*: ¬ *has-disc disc1 m*
and *nm*: *normalized-nnf-match m*

and *am'*: $m' \in set\ (normalize\text{-}primitive\text{-}extract\ (disc,\ sel)\ C'\ f'\ m)$
and *wfdiscsel*: *wf-disc-sel* $(disc,sel)\ C'$
and *disc-different*: $\forall\, a.\ \neg\ disc1\ (C'\ a)$

from *disc-different* **have** *af*: $\forall\, spts.\ (\forall\, a \in Match\ `\ C'\ `\ set\ (f'\ spts).\ \neg$ *has-disc disc1 a*)
by(*simp*)

obtain *as ms* **where** *asms*: *primitive-extractor* $(disc,\ sel)\ m = (as,\ ms)$ **by** *fastforce*

from *am' asms* **have** $m' \in (\lambda spt.\ MatchAnd\ (Match\ (C'\ spt))\ ms)\ `\ set\ (f'$ *as*)
unfolding *normalize-primitive-extract-def* **by**(*simp*)
hence *goalrule*:$\forall\, spt \in set\ (f'\ as).\ \neg$ *has-disc disc1* $(Match\ (C'\ spt)) \Longrightarrow \neg$ *has-disc disc1 ms* $\Longrightarrow \neg$ *has-disc disc1 m'* **by** *fastforce*

from *am primitive-extractor-correct*(*4*)[*OF nm wfdiscsel asms*] **have** *1*: $\neg$ *has-disc disc1 ms* **by** *simp*
from *af* **have** *2*: $\forall\, spt \in set\ (f'\ as).\ \neg$ *has-disc disc1* $(Match\ (C'\ spt))$ **by** *simp*

from *goalrule*[*OF 2 1*] **have** $\neg$ *has-disc disc1 m'* **.**
moreover from *nm* **have** *normalized-nnf-match m'* **by** (*metis am' normalize-primitive-extract-preserves-nnf-normalized wfdiscsel*)
ultimately have $\neg$ *has-disc disc1 m'* $\wedge$ *normalized-nnf-match m'* **by** *simp*
}
hence *x*: $\bigwedge disc\ sel\ C'\ f'.$ *wf-disc-sel* $(disc,\ sel)\ C' \Longrightarrow \forall\, a.\ \neg\ disc1\ (C'\ a) \Longrightarrow$
$\forall\, m.\ normalized\text{-}nnf\text{-}match\ m \wedge \neg\ has\text{-}disc\ disc1\ m \longrightarrow$
$(\forall\, m' \in set\ (normalize\text{-}primitive\text{-}extract\ (disc,\ sel)\ C'\ f'\ m).\ normalized\text{-}nnf\text{-}match\ m' \wedge \neg\ has\text{-}disc\ disc1\ m')$
by *blast*

— Pushing through properties through the ports normalizer
from *normalize-src-ports-preserves-normalized-not-has-disc normalize-src-ports-nnf* **have** *x-src-ports*:
$\forall\, a.\ \neg\ disc\ (Src\text{-}Ports\ a) \Longrightarrow\ \forall\, a.\ \neg\ disc\ (Prot\ a) \Longrightarrow$
$m' \in set\ (normalize\text{-}src\text{-}ports\ m) \Longrightarrow$
$normalized\text{-}nnf\text{-}match\ m \Longrightarrow \neg\ has\text{-}disc\ disc\ m \Longrightarrow \neg\ has\text{-}disc\ disc\ m' \wedge normalized\text{-}nnf\text{-}match\ m'$
for *m m'* **and** *disc* :: *'i common-primitive* $\Rightarrow$ *bool* **by** *blast*
from *normalize-dst-ports-preserves-normalized-not-has-disc normalize-dst-ports-nnf* **have** *x-dst-ports*:
$\forall\, a.\ \neg\ disc\ (Dst\text{-}Ports\ a) \Longrightarrow\ \forall\, a.\ \neg\ disc\ (Prot\ a) \Longrightarrow$
$m' \in set\ (normalize\text{-}dst\text{-}ports\ m) \Longrightarrow$
$normalized\text{-}nnf\text{-}match\ m \Longrightarrow \neg\ has\text{-}disc\ disc\ m \Longrightarrow \neg\ has\text{-}disc\ disc\ m' \wedge normalized\text{-}nnf\text{-}match\ m'$

for $m\ m'$ **and** $disc :: 'i\ common\text{-}primitive \Rightarrow bool$ **by** *blast*

{ fix *rs*
assume $(\forall a.\ \neg\ disc1\ (IIface\ a)) \vee disc1 = is\text{-}Iiface$
and $((\forall a.\ \neg\ disc1\ (OIface\ a)) \vee disc1 = is\text{-}Oiface)$
and $(\forall a.\ \neg\ disc1\ (Prot\ a))$
hence $\forall m{\in}set\ rs.\ \neg\ has\text{-}disc\ disc1\ (get\text{-}match\ m) \wedge normalized\text{-}nnf\text{-}match\ (get\text{-}match\ m) \Longrightarrow$
$\forall m{\in}set\ (optimize\text{-}matches\text{-}option\ compress\text{-}normalize\text{-}besteffort\ rs).$
$normalized\text{-}nnf\text{-}match\ (get\text{-}match\ m) \wedge \neg\ has\text{-}disc\ disc1\ (get\text{-}match\ m)$
apply −
apply(*rule optimize-matches-option-preserves*)
apply(*elim disjE*)
using *compress-normalize-besteffort-hasdisc* **apply** *blast*
using *compress-normalize-besteffort-nnf compress-normalize-besteffort-not-introduces-Iiface compress-normalize-besteffort-not-introduces-Oiface* **by** *blast+*
} note *y=this*

have $\forall a.\ \neg\ disc1\ (Src\text{-}Ports\ a) \Longrightarrow \forall a.\ \neg\ disc1\ (Dst\text{-}Ports\ a) \Longrightarrow$
$\forall a.\ \neg\ disc1\ (Src\ a) \Longrightarrow \forall a.\ \neg\ disc1\ (Dst\ a) \Longrightarrow$
$(\forall a.\ \neg\ disc1\ (IIface\ a)) \vee disc1 = is\text{-}Iiface \Longrightarrow$
$(\forall a.\ \neg\ disc1\ (OIface\ a)) \vee disc1 = is\text{-}Oiface \Longrightarrow (\forall a.\ \neg\ disc1\ (Prot\ a)) \Longrightarrow$
$\forall\ r{\in}set\ rs.\ \neg\ has\text{-}disc\ disc1\ (get\text{-}match\ r) \wedge normalized\text{-}nnf\text{-}match\ (get\text{-}match\ r) \Longrightarrow$
$\forall\ r \in set\ (transform\text{-}normalize\text{-}primitives\ rs).\ normalized\text{-}nnf\text{-}match\ (get\text{-}match\ r) \wedge \neg\ has\text{-}disc\ disc1\ (get\text{-}match\ r)$
unfolding *transform-normalize-primitives-def*
apply(*simp*)
apply(*rule y*)
apply(*simp*; *fail*)
apply(*simp*; *fail*)
apply(*simp*; *fail*)
apply(*rule normalize-rules-preserves*)+
apply(*simp*; *fail*)
subgoal
apply(*intro allI impI conjI ballI*)
apply(*rule rewrite-MultiportPorts-preserves-normalized-not-has-disc*, *simp-all*)
by(*simp add*: *rewrite-MultiportPorts-normalized-nnf-match*)
subgoal
apply *clarify*
apply(*rule x-src-ports*)
by *simp+*
subgoal
apply *clarify*
by(*rule x-dst-ports*) *simp+*
using *x*[*OF wf-disc-sel-common-primitive*(*3*), *of ipt-iprange-compress*,*folded normalize-src-ips-def*] **apply** *blast*

using *x*[*OF wf-disc-sel-common-primitive*(*4*), *of ipt-iprange-compress*,*folded normalize-dst-ips-def*] **apply** *blast*
done

thus *unchanged disc1* $\Longrightarrow$ *changeddisc disc1* $\Longrightarrow \forall\, a.\ \neg$ *disc1* (*Prot a*) $\Longrightarrow$
$\forall\ r \in$ *set rs*. $\neg$ *has-disc disc1* (*get-match r*) $\Longrightarrow$
$\forall\ r \in$ *set* (*transform-normalize-primitives rs*). $\neg$ *has-disc disc1* (*get-match r*)
unfolding *unchanged-def changeddisc-def* **using** *normalized* **by** *blast*

{ **fix** *m* **and** *m'* **and** *disc*::('*i*::*len common-primitive* $\Rightarrow$ *bool*)
and *sel*::('*i*::*len common-primitive* $\Rightarrow$ '*x*) **and** *C'*:: ('*x* $\Rightarrow$ '*i*::*len common-primitive*)
and *f'*::('*x negation-type list* $\Rightarrow$ '*x list*) **and** *neg*
and *disc3*
assume *am*: $\neg$ *has-disc-negated disc3 neg m*
and *nm*: *normalized-nnf-match m*
and *am'*: *m'* $\in$ *set* (*normalize-primitive-extract* (*disc*, *sel*) *C' f' m*)
and *wfdiscsel*: *wf-disc-sel* (*disc*,*sel*) *C'*
and *disc-different*: $\forall\, a.\ \neg$ *disc3* (*C' a*)

from *disc-different* **have** *af*: $\forall\, spts.$ ($\forall\, a \in$ *Match* ' *C'* ' *set* (*f' spts*). $\neg$ *has-disc disc3 a*)
by(*simp*)

obtain *as ms* **where** *asms*: *primitive-extractor* (*disc*, *sel*) *m* = (*as*, *ms*) **by** *fastforce*

from *am' asms* **have** *m'* $\in$ (λ*spt*. *MatchAnd* (*Match* (*C' spt*)) *ms*) ' *set* (*f' as*)
unfolding *normalize-primitive-extract-def* **by**(*simp*)
hence *goalrule*:$\forall\, spt \in$ *set* (*f' as*). $\neg$ *has-disc-negated disc3 neg* (*Match* (*C' spt*)) $\Longrightarrow$
$\neg$ *has-disc-negated disc3 neg ms* $\Longrightarrow$ $\neg$ *has-disc-negated disc3 neg m'* **by** *fastforce*

from *am primitive-extractor-correct*(*6*)[*OF nm wfdiscsel asms*] **have** *1*: $\neg$ *has-disc-negated disc3 neg ms* **by** *simp*
from *af* **have** *2*: $\forall\, spt \in$ *set* (*f' as*). $\neg$ *has-disc-negated disc3 neg* (*Match* (*C' spt*)) **by** *simp*

from *goalrule*[*OF 2 1*] **have** $\neg$ *has-disc-negated disc3 neg m'* .
moreover from *nm* **have** *normalized-nnf-match m'* **by** (*metis am' normalize-primitive-extract-preserves-nnf-normalized wfdiscsel*)
ultimately have $\neg$ *has-disc-negated disc3 neg m'* $\wedge$ *normalized-nnf-match m'* **by** *simp*
}
note *x-generic*=*this*
hence *x*: *wf-disc-sel* (*disc*, *sel*) *C'* $\Longrightarrow \forall\, a.\ \neg$ *disc3* (*C' a*) $\Longrightarrow$

$\forall\, m.\ normalized\text{-}nnf\text{-}match\ m \wedge \neg\ has\text{-}disc\text{-}negated\ disc3\ False\ m \longrightarrow$
$(\forall\, m' \in set\ (normalize\text{-}primitive\text{-}extract\ (disc,\ sel)\ C'\ f'\ m).$
$normalized\text{-}nnf\text{-}match\ m' \wedge \neg\ has\text{-}disc\text{-}negated\ disc3\ False\ m')$
for *disc* :: *'i common-primitive* ⇒ *bool* **and** *sel* **and** *C'* :: *'c* ⇒ *'i common-primitive* **and** *f'* **and** *disc3*
by *blast*

— Pushing through properties through the ports normalizer
from *normalize-src-ports-preserves-normalized-not-has-disc-negated normalize-src-ports-nnf*
have *x-src-ports*:
$\forall\, a.\ \neg\ disc\ (Src\text{-}Ports\ a) \Longrightarrow (\forall\, a.\ \neg\ disc\ (Prot\ a)) \vee \neg\ has\text{-}disc\text{-}negated\ is\text{-}Src\text{-}Ports\ False\ m \Longrightarrow$
$m' \in set\ (normalize\text{-}src\text{-}ports\ m) \Longrightarrow$
$normalized\text{-}nnf\text{-}match\ m \Longrightarrow \neg\ has\text{-}disc\text{-}negated\ disc\ False\ m \Longrightarrow$
$\neg\ has\text{-}disc\text{-}negated\ disc\ False\ m' \wedge normalized\text{-}nnf\text{-}match\ m'$
for *m m'* **and** *disc* :: *'i common-primitive* ⇒ *bool* **by** *blast*

from *normalize-dst-ports-preserves-normalized-not-has-disc-negated normalize-dst-ports-nnf*
have *x-dst-ports*:
$\forall\, a.\ \neg\ disc\ (Src\text{-}Ports\ a) \Longrightarrow (\forall\, a.\ \neg\ disc\ (Prot\ a)) \vee \neg\ has\text{-}disc\text{-}negated\ is\text{-}Dst\text{-}Ports\ False\ m \Longrightarrow$
$m' \in set\ (normalize\text{-}dst\text{-}ports\ m) \Longrightarrow$
$normalized\text{-}nnf\text{-}match\ m \Longrightarrow \neg\ has\text{-}disc\text{-}negated\ disc\ False\ m \Longrightarrow$
$\neg\ has\text{-}disc\text{-}negated\ disc\ False\ m' \wedge normalized\text{-}nnf\text{-}match\ m'$
for *m m'* **and** *disc* :: *'i common-primitive* ⇒ *bool* **by** *blast*

{ fix *rs*
fix *P* :: *'i common-primitive match-expr* ⇒ *bool*
assume $(\forall\, a.\ \neg\ disc3\ (IIface\ a)) \vee disc3 = is\text{-}Iiface$
and $(\forall\, a.\ \neg\ disc3\ (OIface\ a)) \vee disc3 = is\text{-}Oiface$
and $(\forall\, a.\ \neg\ disc3\ (Prot\ a)) \vee disc3 = is\text{-}Prot$
and *P-prop*: $\forall\, m\ m'.\ normalized\text{-}nnf\text{-}match\ m \longrightarrow P\ m \longrightarrow compress\text{-}normalize\text{-}besteffort\ m = Some\ m' \longrightarrow P\ m'$
hence
$\forall\, r{\in}set\ rs.\ \neg\ has\text{-}disc\text{-}negated\ disc3\ False\ (get\text{-}match\ r) \wedge normalized\text{-}nnf\text{-}match\ (get\text{-}match\ r) \wedge P\ (get\text{-}match\ r) \Longrightarrow$
$\forall\, r{\in}set\ (optimize\text{-}matches\text{-}option\ compress\text{-}normalize\text{-}besteffort\ rs).$
$normalized\text{-}nnf\text{-}match\ (get\text{-}match\ r) \wedge \neg\ has\text{-}disc\text{-}negated\ disc3\ False\ (get\text{-}match\ r) \wedge P\ (get\text{-}match\ r)$
apply −
thm *optimize-matches-option-preserves*[**where** *P*=
$\lambda m.\ normalized\text{-}nnf\text{-}match\ m \wedge \neg\ has\text{-}disc\text{-}negated\ disc3\ False\ m \wedge P\ m$]
apply(*rule optimize-matches-option-preserves*[**where** *P*=
$\lambda m.\ normalized\text{-}nnf\text{-}match\ m \wedge \neg\ has\text{-}disc\text{-}negated\ disc3\ False\ m \wedge P\ m$])
apply(*elim disjE*)
using *compress-normalize-besteffort-nnf compress-normalize-besteffort-hasdisc-negated*
apply *blast*
using *compress-normalize-besteffort-nnf*

compress-normalize-besteffort-not-introduces-Iiface-negated
compress-normalize-besteffort-not-introduces-Oiface-negated
compress-normalize-besteffort-not-introduces-Prot-negated **by** *blast+*
} **note** *y-generic=this*

note *y=y-generic*[*of* λ*-. True*, *simplified*]

have *case-disc3-is-not-prot*: ∀ *a*. ¬ *disc3* (*Src-Ports a*) ⟹ ∀ *a*. ¬ *disc3* (*Dst-Ports a*) ⟹
∀ *a*. ¬ *disc3* (*Src a*) ⟹ ∀ *a*. ¬ *disc3* (*Dst a*) ⟹
(∀ *a*. ¬ *disc3* (*IIface a*)) ∨ *disc3* = *is-Iiface* ⟹
(∀ *a*. ¬ *disc3* (*OIface a*)) ∨ *disc3* = *is-Oiface* ⟹
(∀ *a*. ¬ *disc3* (*Prot a*)) ⟹
∀ *r* ∈ *set rs*. ¬ *has-disc-negated disc3 False* (*get-match r*) ∧ *normalized-nnf-match* (*get-match r*) ⟹
∀ *r* ∈ *set* (*transform-normalize-primitives rs*). *normalized-nnf-match* (*get-match r*) ∧ ¬ *has-disc-negated disc3 False* (*get-match r*)
unfolding *transform-normalize-primitives-def*
apply(*simp*)
apply(*rule y*)
apply(*simp*; *fail*)
apply(*simp*; *fail*)
apply(*blast*)
apply(*rule normalize-rules-preserves*)+
apply(*simp*; *fail*)
subgoal
apply(*intro allI impI conjI ballI*)
apply(*rule rewrite-MultiportPorts-preserves-normalized-not-has-disc-negated*, *simp-all*)
by(*simp add*: *rewrite-MultiportPorts-normalized-nnf-match*)
subgoal
apply(*clarify*)
apply(*rule-tac m6=m* **in** *x-src-ports*)
by(*simp*)+
subgoal
apply(*clarify*)
by(*rule x-dst-ports*) *simp+*
using *x*[*OF wf-disc-sel-common-primitive*(*3*), *of disc3 ipt-iprange-compress*, *folded normalize-src-ips-def*] **apply** *blast*
using *x*[*OF wf-disc-sel-common-primitive*(*4*), *of disc3 ipt-iprange-compress*, *folded normalize-dst-ips-def*] **apply** *blast*
done

have *case-disc3-is-prot-optimize-matches-option*:∀ *r*∈*set rs*.
¬ *has-disc-negated is-Prot False* (*get-match r*) ∧
normalized-nnf-match (*get-match r*) ∧
¬ *has-disc-negated is-Src-Ports False* (*get-match r*) ∧
¬ *has-disc-negated is-Dst-Ports False* (*get-match r*) ⟹

$\forall r{\in}set\ (optimize\text{-}matches\text{-}option\ compress\text{-}normalize\text{-}besteffort\ rs).$
$normalized\text{-}nnf\text{-}match\ (get\text{-}match\ r) \wedge$
$\neg\ has\text{-}disc\text{-}negated\ is\text{-}Prot\ False\ (get\text{-}match\ r) \wedge$
$\neg\ has\text{-}disc\text{-}negated\ is\text{-}Src\text{-}Ports\ False\ (get\text{-}match\ r) \wedge$
$\neg\ has\text{-}disc\text{-}negated\ is\text{-}Dst\text{-}Ports\ False\ (get\text{-}match\ r)$
if *isprot*: *disc3 = is-Prot*
for *rs* :: *'i common-primitive rule list*
apply(*rule y-generic*[**where** *P8*=$\lambda m.\ \neg\ has\text{-}disc\text{-}negated\ is\text{-}Src\text{-}Ports\ False\ m \wedge \neg\ has\text{-}disc\text{-}negated\ is\text{-}Dst\text{-}Ports\ False\ m$, *simplified isprot*])
apply *simp+*
apply(*clarify*)
apply(*intro conjI*)
using *compress-normalize-besteffort-hasdisc-negated*[*of is-Src-Ports*] **apply** *fastforce*
using *compress-normalize-besteffort-hasdisc-negated*[*of is-Dst-Ports*] **apply** *fastforce*
by *simp*

have *case-disc3-is-prot*: *disc3 = is-Prot* $\Longrightarrow$
$\forall\ r \in set\ rs.\ \neg\ has\text{-}disc\text{-}negated\ disc3\ False\ (get\text{-}match\ r) \wedge normalized\text{-}nnf\text{-}match\ (get\text{-}match\ r) \wedge$
$\neg\ has\text{-}disc\text{-}negated\ is\text{-}Src\text{-}Ports\ False\ (get\text{-}match\ r) \wedge \neg\ has\text{-}disc\text{-}negated\ is\text{-}Dst\text{-}Ports\ False\ (get\text{-}match\ r)\ \&$
$\neg\ has\text{-}disc\ is\text{-}MultiportPorts\ (get\text{-}match\ r)$ — MultiportPorts could be rewritten to negated *Src/Dst* Ports $\Longrightarrow$
$\forall\ r \in set\ (transform\text{-}normalize\text{-}primitives\ rs).\ normalized\text{-}nnf\text{-}match\ (get\text{-}match\ r) \wedge \neg\ has\text{-}disc\text{-}negated\ disc3\ False\ (get\text{-}match\ r) \wedge$
$\neg\ has\text{-}disc\text{-}negated\ is\text{-}Src\text{-}Ports\ False\ (get\text{-}match\ r) \wedge \neg\ has\text{-}disc\text{-}negated\ is\text{-}Dst\text{-}Ports\ False\ (get\text{-}match\ r)$
unfolding *transform-normalize-primitives-def*
apply(*simp*)
apply(*rule case-disc3-is-prot-optimize-matches-option*)
apply(*simp*; *fail*)
thm *normalize-rules-property*[
where *P*=$\lambda m.\ normalized\text{-}nnf\text{-}match\ m \wedge \neg\ has\text{-}disc\text{-}negated\ disc3\ False\ m$]
apply(*rule normalize-rules-property*[
where *P*=$\lambda m.\ normalized\text{-}nnf\text{-}match\ m \wedge$
$\neg\ has\text{-}disc\text{-}negated\ disc3\ False\ m \wedge$
$\neg\ has\text{-}disc\text{-}negated\ is\text{-}Src\text{-}Ports\ False\ m \wedge$
$\neg\ has\text{-}disc\text{-}negated\ is\text{-}Dst\text{-}Ports\ False\ m$])
apply(*rule normalize-rules-property*[
where *P*=$\lambda m.\ normalized\text{-}nnf\text{-}match\ m \wedge$
$\neg\ has\text{-}disc\text{-}negated\ disc3\ False\ m \wedge$
$\neg\ has\text{-}disc\text{-}negated\ is\text{-}Src\text{-}Ports\ False\ m \wedge$
$\neg\ has\text{-}disc\text{-}negated\ is\text{-}Dst\text{-}Ports\ False\ m$])
apply(*rule normalize-rules-property*[
where *P*=$\lambda m.\ normalized\text{-}nnf\text{-}match\ m \wedge$

¬ *has-disc-negated disc3 False m* ∧
¬ *has-disc-negated is-Src-Ports False m* ∧
¬ *has-disc-negated is-Dst-Ports False m*])
apply(*rule normalize-rules-property*[
where *P*=λ*m. normalized-nnf-match m* ∧
¬ *has-disc-negated disc3 False m* ∧
¬ *has-disc-negated is-Src-Ports False m* ∧
¬ *has-disc-negated is-Dst-Ports False m*])
apply(*rule normalize-rules-property*[
where *P*=λ*m. normalized-nnf-match m* ∧
¬ *has-disc-negated disc3 False m* ∧
¬ *has-disc-negated is-Src-Ports False m* ∧
¬ *has-disc-negated is-Dst-Ports False m* ∧
¬ *has-disc is-MultiportPorts m*])
apply(*simp*; *fail*)
subgoal
apply(*intro allI impI conjI ballI*)
apply(*simp add*: *rewrite-MultiportPorts-normalized-nnf-match*; *fail*)
apply(*rule rewrite-MultiportPorts-preserves-normalized-not-has-disc-negated*, *simp-all*)
— Here we need ¬ *has-disc is-MultiportPorts m*
using *rewrite-MultiportPorts-unchanged-if-not-has-disc* **by** *fastforce*+
subgoal
apply(*clarify*)
apply(*frule-tac m6*=*m* **in** *x-src-ports*[*rotated 2*])
apply(*simp-all*)
apply *simp*
using *normalize-src-ports-preserves-normalized-not-has-disc-negated* **by** *blast*
subgoal
apply(*clarify*)
apply(*frule-tac m6*=*m* **in** *x-dst-ports*[*rotated 2*])
apply(*simp-all*)
apply *simp*
using *normalize-dst-ports-preserves-normalized-not-has-disc-negated* **by** *blast*
using *x*[*OF wf-disc-sel-common-primitive*(*3*), *of disc3 ipt-iprange-compress*, *folded normalize-src-ips-def*]
x[*OF wf-disc-sel-common-primitive*(*3*), *of is-Dst-Ports ipt-iprange-compress*, *folded normalize-src-ips-def*]
x-generic[*OF - - - wf-disc-sel-common-primitive*(*3*), *of is-Src-Ports False - - ipt-iprange-compress*, *folded normalize-src-ips-def*]
apply (*meson common-primitive.disc*(*45*) *common-primitive.disc*(*56*) *common-primitive.disc*(*67*); *fail*)
using *x*[*OF wf-disc-sel-common-primitive*(*4*), *of disc3 ipt-iprange-compress*, *folded normalize-dst-ips-def*]
x[*OF wf-disc-sel-common-primitive*(*4*), *of is-Src-Ports ipt-iprange-compress*, *folded normalize-dst-ips-def*]
x-generic[*OF - - - wf-disc-sel-common-primitive*(*4*), *of is-Dst-Ports False - - ipt-iprange-compress*, *folded normalize-dst-ips-def*]
apply (*meson common-primitive.disc*(*46*) *common-primitive.disc*(*57*)

common-primitive.disc(*68*); *fail*)
done

show *unchanged disc3* ⟹ *changeddisc disc3* ⟹
(∀ *a*. ¬ *disc3* (*Prot a*)) ∨
(*disc3* = *is-Prot* ∧ (∀ *r* ∈ *set rs*.
¬ *has-disc-negated is-Src-Ports False* (*get-match r*) ∧
¬ *has-disc-negated is-Dst-Ports False* (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*))) ⟹
∀ *r* ∈ *set rs*. ¬ *has-disc-negated disc3 False* (*get-match r*) ⟹
∀ *r* ∈ *set* (*transform-normalize-primitives rs*). ¬ *has-disc-negated disc3 False* (*get-match r*)
unfolding *unchanged-def changeddisc-def*
apply(*elim disjE*)
using *normalized case-disc3-is-not-prot* **apply** *blast*
using *normalized case-disc3-is-prot* **by** *blast*
qed

theorem *iiface-constrain*:
assumes *simplers*: *simple-ruleset rs*
and *normalized*: ∀ *r* ∈ *set rs*. *normalized-nnf-match* (*get-match r*)
and *wf-ipassmt*: *ipassmt-sanity-nowildcards ipassmt*
and *nospoofing*: ⋀*ips*. *ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* ⟹ *p-src p* ∈ *ipcidr-union-set* (*set ips*)
shows (*common-matcher*, α),*p*⊢ ⟨*optimize-matches* (*iiface-constrain ipassmt*) *rs*, *s*⟩ ⇒α *t* ⟷ (*common-matcher*, α),*p*⊢ ⟨*rs*, *s*⟩ ⇒α *t*
and *simple-ruleset* (*optimize-matches* (*iiface-constrain ipassmt*) *rs*)
proof −
show *simplers-t*: *simple-ruleset* (*optimize-matches* (*iiface-constrain ipassmt*) *rs*)
by (*simp add*: *optimize-matches-simple-ruleset simplers*)

have *my-arg-cong*: ⋀*P Q*. *P s* = *Q s* ⟹ (*P s* = *t*) ⟷ (*Q s* = *t*) **by** *simp*

show (*common-matcher*, α),*p*⊢ ⟨*optimize-matches* (*iiface-constrain ipassmt*) *rs*, *s*⟩ ⇒α *t* ⟷ (*common-matcher*, α),*p*⊢ ⟨*rs*, *s*⟩ ⇒α *t*
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers-t*]]
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers*]]
apply(*rule my-arg-cong*)
apply(*rule optimize-matches-generic*[**where** *P*=λ *m* -. *normalized-nnf-match m*])
apply(*simp add*: *normalized*)
apply(*rule matches-iiface-constrain*)
apply(*simp-all add*: *wf-ipassmt nospoofing*)
done
qed

In contrast to ⟦*simple-ruleset ?rs*; ∀ *r*∈*set ?rs*. *normalized-nnf-match* (*get-match*

r); *ipassmt-sanity-nowildcards ?ipassmt*; $\bigwedge$*ips. ?ipassmt* (*Iface* (*p-iiface ?p*)) = *Some ips* $\Longrightarrow$ *p-src ?p* $\in$ *ipcidr-union-set* (*set ips*)$\rrbracket$ $\Longrightarrow$ (*common-matcher*, *?α*),*?p*$\vdash$ $\langle$*optimize-matches* (*iiface-constrain ?ipassmt*) *?rs*, *?s*$\rangle$ $\Rightarrow_\alpha$ *?t* = (*common-matcher*, *?α*),*?p*$\vdash$ $\langle$*?rs*, *?s*$\rangle$ $\Rightarrow_\alpha$ *?t*

$\llbracket$*simple-ruleset ?rs*; $\forall$ *r*$\in$*set ?rs. normalized-nnf-match* (*get-match r*); *ipassmt-sanity-nowildcards ?ipassmt*; $\bigwedge$*ips. ?ipassmt* (*Iface* (*p-iiface ?p*)) = *Some ips* $\Longrightarrow$ *p-src ?p* $\in$ *ipcidr-union-set* (*set ips*)$\rrbracket$ $\Longrightarrow$ *simple-ruleset* (*optimize-matches* (*iiface-constrain ?ipassmt*) *?rs*), this requires *ipassmt-sanity-disjoint* and as much stronger nospoof assumption: This assumption requires that the packet is actually in ipassmt!

theorem *iiface-rewrite*:
assumes *simplers*: *simple-ruleset rs*
and *normalized*: $\forall$ *r* $\in$ *set rs. normalized-nnf-match* (*get-match r*)
and *wf-ipassmt*: *ipassmt-sanity-nowildcards ipassmt*
and *disjoint-ipassmt*: *ipassmt-sanity-disjoint ipassmt*
and *nospoofing*: $\exists$ *ips. ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* $\wedge$ *p-src p* $\in$ *ipcidr-union-set* (*set ips*)
shows (*common-matcher*, *α*),*p*$\vdash$ $\langle$*optimize-matches* (*iiface-rewrite ipassmt*) *rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t* $\longleftrightarrow$ (*common-matcher*, *α*),*p*$\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t*
and *simple-ruleset* (*optimize-matches* (*iiface-rewrite ipassmt*) *rs*)
proof −
show *simplers-t*: *simple-ruleset* (*optimize-matches* (*iiface-rewrite ipassmt*) *rs*)
by(*simp add*: *simplers optimize-matches-simple-ruleset*)

— packet must come from a defined interface!
from *nospoofing* **have** *Iface* (*p-iiface p*) $\in$ *dom ipassmt* **by** *blast*

have *my-arg-cong*: $\bigwedge$*P Q. P s* = *Q s* $\Longrightarrow$ (*P s* = *t*) $\longleftrightarrow$ (*Q s* = *t*) **by** *simp*

show (*common-matcher*, *α*),*p*$\vdash$ $\langle$*optimize-matches* (*iiface-rewrite ipassmt*) *rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t* $\longleftrightarrow$ (*common-matcher*, *α*),*p*$\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t*
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers-t*]]
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers*]]
apply(*rule my-arg-cong*)
apply(*rule optimize-matches-generic*[**where** *P*=λ *m* -. *normalized-nnf-match m*])
apply(*simp add*: *normalized*)
apply(*rule matches-iiface-rewrite*)
apply(*simp-all add*: *wf-ipassmt nospoofing disjoint-ipassmt*)
done
qed

theorem *oiface-rewrite*:
assumes *simplers*: *simple-ruleset rs*
and *normalized*: $\forall$ *r* $\in$ *set rs. normalized-nnf-match* (*get-match r*)

and *wf-ipassmt*: *ipassmt-sanity-nowildcards ipassmt*
and *ipassmt-from-rt*: *ipassmt = map-of* (*routing-ipassmt rt*)
and *correct-routing*: *correct-routing rt*
and *rtbl-decided*: *output-iface* (*routing-table-semantics rt* (*p-dst p*)) = *p-oiface p*
shows (*common-matcher*, *α*),*p*⊢ ⟨*optimize-matches* (*oiface-rewrite ipassmt*) *rs*, *s*⟩ ⇒$_\alpha$ *t* ⟷ (*common-matcher*, *α*),*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t*
and *simple-ruleset* (*optimize-matches* (*oiface-rewrite ipassmt*) *rs*)
proof −
show *simplers-t*: *simple-ruleset* (*optimize-matches* (*oiface-rewrite ipassmt*) *rs*)
using *simplers* **by**(*fact optimize-matches-simple-ruleset*)
show (*common-matcher*, *α*),*p*⊢ ⟨*optimize-matches* (*oiface-rewrite ipassmt*) *rs*, *s*⟩ ⇒$_\alpha$ *t* ⟷ (*common-matcher*, *α*),*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t*
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers-t*]]
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers*]]
apply(*rule arg-cong*[**where** *f*=λ*x*. *x* = *t*])
apply(*rule optimize-matches-generic*[**where** *P*=λ *m* -. *normalized-nnf-match m*])
apply(*simp add*: *normalized* ;*fail*)
apply(*rule matches-oiface-rewrite*[*OF* - - - *ipassmt-from-rt*]; *assumption?*)
apply(*simp-all add*: *wf-ipassmt correct-routing rtbl-decided*)
done
qed

definition *upper-closure* :: *'i*::*len common-primitive rule list* ⇒ *'i common-primitive rule list* **where**
upper-closure rs == *remdups-rev* (*transform-optimize-dnf-strict*
(*transform-normalize-primitives* (*transform-optimize-dnf-strict* (*optimize-matches-a upper-closure-matchexpr rs*))))
definition *lower-closure* :: *'i*::*len common-primitive rule list* ⇒ *'i common-primitive rule list* **where**
lower-closure rs == *remdups-rev* (*transform-optimize-dnf-strict*
(*transform-normalize-primitives* (*transform-optimize-dnf-strict* (*optimize-matches-a lower-closure-matchexpr rs*))))

putting it all together

lemma *transform-upper-closure*:
assumes *simplers*: *simple-ruleset rs*
— semantics are preserved
shows (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*upper-closure rs*, *s*⟩ ⇒$_\alpha$ *t* ⟷ (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t*
and *simple-ruleset* (*upper-closure rs*)
— simple, normalized rules without unknowns
and ∀ *r* ∈ *set* (*upper-closure rs*). *normalized-nnf-match* (*get-match r*) ∧
normalized-src-ports (*get-match r*) ∧
normalized-dst-ports (*get-match r*) ∧

normalized-src-ips (*get-match r*) ∧
normalized-dst-ips (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*) ∧
¬ *has-disc is-Extra* (*get-match r*)
— no new primitives are introduced
and ∀ *a*. ¬ *disc* (*Src-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Dst-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Src a*) ⟹ ∀ *a*. ¬ *disc* (*Dst a*) ⟹
∀ *a*. ¬ *disc* (*IIface a*) ∨ *disc* = *is-Iiface* ⟹ ∀ *a*. ¬ *disc* (*OIface a*) ∨ *disc* = *is-Oiface* ⟹
∀ *a*. ¬ *disc* (*Prot a*) ⟹
∀ *r* ∈ *set rs*. ¬ *has-disc disc* (*get-match r*) ⟹ ∀ *r* ∈ *set* (*upper-closure rs*). ¬ *has-disc disc* (*get-match r*)
and ∀ *a*. ¬ *disc* (*Src-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Dst-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Src a*) ⟹ ∀ *a*. ¬ *disc* (*Dst a*) ⟹
∀ *a*. ¬ *disc* (*IIface a*) ∨ *disc* = *is-Iiface* ⟹ ∀ *a*. ¬ *disc* (*OIface a*) ∨ *disc* = *is-Oiface* ⟹
(∀ *a*. ¬ *disc* (*Prot a*)) ∨
disc = *is-Prot* ∧ — if it is prot, there must not be negated matches on ports
(∀ *r* ∈ *set rs*. ¬ *has-disc-negated is-Src-Ports False* (*get-match r*) ∧
¬ *has-disc-negated is-Dst-Ports False* (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*)) ⟹
∀ *r* ∈ *set rs*. ¬ *has-disc-negated disc False* (*get-match r*) ⟹
∀ *r* ∈ *set* (*upper-closure rs*). ¬ *has-disc-negated disc False* (*get-match r*)
proof −
let *?rs1*=*optimize-matches-a upper-closure-matchexpr rs*
let *?rs2*=*transform-optimize-dnf-strict ?rs1*
let *?rs3*=*transform-normalize-primitives ?rs2*
let *?rs4*=*transform-optimize-dnf-strict ?rs3*

{ fix *m a*
have *Rule m a* ∈ *set* (*upper-closure rs*) ⟹
(*a* = *action.Accept* ∨ *a* = *action.Drop*) ∧
normalized-nnf-match m ∧
normalized-src-ports m ∧
normalized-dst-ports m ∧
normalized-src-ips m ∧
normalized-dst-ips m ∧
¬ *has-disc is-MultiportPorts m* ∧
¬ *has-disc is-Extra m*
using *simplers*
unfolding *upper-closure-def*
apply(*simp add*: *remdups-rev-set*)
apply(*frule transform-remove-unknowns-upper*(*4*))
apply(*drule transform-remove-unknowns-upper*(*2*))
thm *transform-optimize-dnf-strict*[*OF - wf-in-doubt-allow*]
apply(*frule*(*1*) *transform-optimize-dnf-strict-structure*(*2*)[*OF - wf-in-doubt-allow*, **where** *disc*=*is-Extra*])
apply(*thin-tac* ∀ *r*∈ *set* (*optimize-matches-a upper-closure-matchexpr rs*). ¬ *has-disc is-Extra* (*get-match r*))

apply(*frule transform-optimize-dnf-strict-structure*(*3*)[*OF - wf-in-doubt-allow*])
apply(*drule transform-optimize-dnf-strict-structure*(*1*)[*OF - wf-in-doubt-allow*])
thm *transform-normalize-primitives*[*OF - wf-in-doubt-allow*]
apply(*frule*(*1*) *transform-normalize-primitives*(*3*)[*OF - wf-in-doubt-allow, of - is-Extra*])
apply(*simp*;*fail*)
apply(*simp*;*fail*)
apply(*simp*;*fail*)
apply *blast*
apply(*thin-tac* ∀ *r*∈ *set ?rs2*. ¬ *has-disc is-Extra* (*get-match r*))
apply(*frule*(*1*) *transform-normalize-primitives*(*5*)[*OF - wf-in-doubt-allow*])
apply(*drule transform-normalize-primitives*(*2*)[*OF - wf-in-doubt-allow*], *simp*)
thm *transform-optimize-dnf-strict*[*OF - wf-in-doubt-allow*]
apply(*frule*(*1*) *transform-optimize-dnf-strict-structure*(*2*)[*OF - wf-in-doubt-allow,* **where** *disc=is-Extra*])
apply(*frule transform-optimize-dnf-strict-structure*(*2*)[*OF - wf-in-doubt-allow,* **where** *disc=is-MultiportPorts*])
apply *blast*
apply(*frule transform-optimize-dnf-strict-structure*(*3*)[*OF - wf-in-doubt-allow*])
apply(*frule transform-optimize-dnf-strict-structure*(*4*)[*OF - wf-in-doubt-allow, of - (is-Src-Ports, src-ports-sel) (λps. case ps of L4Ports - pts ⇒ length pts ≤ 1)*])
apply(*simp add*: *normalized-src-ports-def2*; *fail*)
apply(*frule transform-optimize-dnf-strict-structure*(*4*)[*OF - wf-in-doubt-allow, of - (is-Dst-Ports, dst-ports-sel) (λps. case ps of L4Ports - pts ⇒ length pts ≤ 1)*])
apply(*simp add*: *normalized-dst-ports-def2*; *fail*)
apply(*frule transform-optimize-dnf-strict-structure*(*4*)[*OF - wf-in-doubt-allow, of - (is-Src, src-sel) normalized-cidr-ip*])
apply(*simp add*: *normalized-src-ips-def2*; *fail*)
apply(*frule transform-optimize-dnf-strict-structure*(*4*)[*OF - wf-in-doubt-allow, of - (is-Dst, dst-sel) normalized-cidr-ip*])
apply(*simp add*: *normalized-dst-ips-def2*; *fail*)
apply(*thin-tac* ∀ *r*∈*set ?rs2*. - *r*)+
apply(*thin-tac* ∀ *r*∈*set ?rs3*. - *r*)+
apply(*drule transform-optimize-dnf-strict-structure*(*1*)[*OF - wf-in-doubt-allow*])
apply(*subgoal-tac* (*a* = *action.Accept* ∨ *a* = *action.Drop*))
prefer *2*
apply(*simp-all add*: *simple-ruleset-def*)
apply *fastforce*
apply(*simp add*: *normalized-src-ports-def2 normalized-dst-ports-def2 normalized-src-ips-def2 normalized-dst-ips-def2*)
apply(*intro conjI*)
by *fastforce*+
} **note** *1*=*this*

from *1* **show** *simple-ruleset* (*upper-closure rs*)
apply(*simp add*: *simple-ruleset-def*)
apply(*clarify*)
apply(*rename-tac r*)

apply(*case-tac r*)
apply(*simp*)
by *blast*

from *1* **show** $\forall\ r \in$ *set* (*upper-closure rs*). *normalized-nnf-match* (*get-match r*) $\wedge$
normalized-src-ports (*get-match r*) $\wedge$
normalized-dst-ports (*get-match r*) $\wedge$
normalized-src-ips (*get-match r*) $\wedge$
normalized-dst-ips (*get-match r*) $\wedge$
$\neg$ *has-disc is-MultiportPorts* (*get-match r*) $\wedge$
$\neg$ *has-disc is-Extra* (*get-match r*)
apply(*clarify*)
apply(*rename-tac r*)
apply(*case-tac r*)
apply(*simp*)
done

show $\forall$ *a*. $\neg$ *disc* (*Src-Ports a*) $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc* (*Dst-Ports a*) $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc* (*Src a*) $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc* (*Dst a*) $\Longrightarrow$
$\forall$ *a*. $\neg$ *disc* (*IIface a*) $\vee$ *disc* = *is-Iiface* $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc* (*OIface a*) $\vee$ *disc* = *is-Oiface* $\Longrightarrow$
$\forall$ *a*. $\neg$ *disc* (*Prot a*) $\Longrightarrow$
$\forall\ r \in$ *set rs*. $\neg$ *has-disc disc* (*get-match r*) $\Longrightarrow$ $\forall\ r \in$ *set* (*upper-closure rs*). $\neg$ *has-disc disc* (*get-match r*)
using *simplers*
unfolding *upper-closure-def*
apply $-$
apply(*frule*(*1*) *transform-remove-unknowns-upper*(*3*)[**where** *disc*=*disc*])
apply(*drule transform-remove-unknowns-upper*(*2*))
apply(*frule*(*1*) *transform-optimize-dnf-strict-structure*(*2*)[*OF - wf-in-doubt-allow*, **where** *disc*=*disc*])
apply(*frule transform-optimize-dnf-strict-structure*(*3*)[*OF - wf-in-doubt-allow*])
apply(*drule transform-optimize-dnf-strict-structure*(*1*)[*OF - wf-in-doubt-allow*])
apply(*frule*(*1*) *transform-normalize-primitives*(*3*)[*OF - wf-in-doubt-allow*, *of - disc*])
apply(*simp*;*fail*)
apply *blast*
apply(*simp*;*fail*)
apply(*simp*;*fail*)
apply(*drule transform-normalize-primitives*(*2*)[*OF - wf-in-doubt-allow*], *simp*)
apply(*frule*(*1*) *transform-optimize-dnf-strict-structure*(*2*)[*OF - wf-in-doubt-allow*, **where** *disc*=*disc*])
apply(*simp add*: *remdups-rev-set*)
done

have *no-ports-1*:

¬ *has-disc-negated is-Src-Ports False* (*get-match m*) ∧
¬ *has-disc-negated is-Dst-Ports False* (*get-match m*) ∧
¬ *has-disc is-MultiportPorts* (*get-match m*)
if *no-ports*: ∀ *r*∈*set rs*.
¬ *has-disc-negated is-Src-Ports False* (*get-match r*) ∧
¬ *has-disc-negated is-Dst-Ports False* (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*)
and *m*: *m* ∈ *set* (*transform-optimize-dnf-strict* (*optimize-matches-a upper-closure-matchexpr rs*))
for *m*
proof −
from *no-ports transform-remove-unknowns-upper*(*3*,*6*)[*OF simplers*] **have**
∀ *r*∈ *set* (*optimize-matches-a upper-closure-matchexpr rs*).
¬ *has-disc-negated is-Src-Ports False* (*get-match r*) ∧
¬ *has-disc-negated is-Dst-Ports False* (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*)
by *blast*
with *m transform-optimize-dnf-strict-structure*(*2*,*5*)[*OF optimize-matches-a-simple-ruleset*[*OF simplers*] *wf-in-doubt-allow*, *of upper-closure-matchexpr*]
show *?thesis* **by** *blast*
qed

show∀ *a*. ¬ *disc* (*Src-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Dst-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Src a*) ⟹ ∀ *a*. ¬ *disc* (*Dst a*) ⟹
∀ *a*. ¬ *disc* (*IIface a*) ∨ *disc* = *is-Iiface* ⟹ ∀ *a*. ¬ *disc* (*OIface a*) ∨ *disc* = *is-Oiface* ⟹
(∀ *a*. ¬ *disc* (*Prot a*)) ∨ *disc* = *is-Prot* ∧
(∀ *r* ∈ *set rs*. ¬ *has-disc-negated is-Src-Ports False* (*get-match r*) ∧
¬ *has-disc-negated is-Dst-Ports False* (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*)) ⟹
∀ *r* ∈ *set rs*. ¬ *has-disc-negated disc False* (*get-match r*) ⟹
∀ *r* ∈ *set* (*upper-closure rs*). ¬ *has-disc-negated disc False* (*get-match r*)
using *simplers*
unfolding *upper-closure-def*
apply −
apply(*frule*(*1*) *transform-remove-unknowns-upper*(*6*)[**where** *disc*=*disc*])
apply(*drule transform-remove-unknowns-upper*(*2*))
apply(*frule*(*1*) *transform-optimize-dnf-strict-structure*(*5*)[*OF* - *wf-in-doubt-allow*, **where** *disc*=*disc*])
apply(*frule transform-optimize-dnf-strict-structure*(*3*)[*OF* - *wf-in-doubt-allow*])
apply(*drule transform-optimize-dnf-strict-structure*(*1*)[*OF* - *wf-in-doubt-allow*])
apply(*frule*(*1*) *transform-normalize-primitives*(*7*)[*OF* - *wf-in-doubt-allow*, *of* - *disc*])
apply(*simp*;*fail*)
apply *blast*
apply(*elim disjE*)
apply *blast*
apply(*simp*)
using *no-ports-1* **apply** *fast*

apply(*simp*;*fail*)
apply(*drule transform-normalize-primitives*(*2*)[*OF - wf-in-doubt-allow*], *simp*)
apply(*frule*(*1*) *transform-optimize-dnf-strict-structure*(*5*)[*OF - wf-in-doubt-allow*, **where** *disc*=*disc*])
apply(*simp add*: *remdups-rev-set*)
done

show (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*upper-closure rs*, *s*⟩ ⇒$_\alpha$ *t* ⟷ (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t*
using *simplers*
unfolding *upper-closure-def*
apply −
apply(*frule transform-remove-unknowns-upper*(*1*)[**where** *p*=*p* **and** *s*=*s* **and** *t*=*t*])
apply(*drule transform-remove-unknowns-upper*(*2*))
apply(*frule transform-optimize-dnf-strict*[*OF - wf-in-doubt-allow*, **where** *p*=*p* **and** *s*=*s* **and** *t*=*t*])
apply(*frule transform-optimize-dnf-strict-structure*(*3*)[*OF - wf-in-doubt-allow*])
apply(*drule transform-optimize-dnf-strict-structure*(*1*)[*OF - wf-in-doubt-allow*])
apply(*frule*(*1*) *transform-normalize-primitives*(*1*)[*OF - wf-in-doubt-allow*, **where** *p*=*p* **and** *s*=*s* **and** *t*=*t*])
apply(*drule transform-normalize-primitives*(*2*)[*OF - wf-in-doubt-allow*], *simp*)
apply(*frule transform-optimize-dnf-strict*[*OF - wf-in-doubt-allow*, **where** *p*=*p* **and** *s*=*s* **and** *t*=*t*])
apply(*drule transform-optimize-dnf-strict-structure*(*1*)[*OF - wf-in-doubt-allow*])
apply(*simp*)
using *approximating-bigstep-fun-remdups-rev*
by (*simp add*: *approximating-bigstep-fun-remdups-rev approximating-semantics-iff-fun-good-ruleset remdups-rev-simplers simple-imp-good-ruleset*)
qed

lemma *transform-lower-closure*:
assumes *simplers*: *simple-ruleset rs*
— semantics are preserved
shows (*common-matcher*, *in-doubt-deny*),*p*⊢ ⟨*lower-closure rs*, *s*⟩ ⇒$_\alpha$ *t* ⟷ (*common-matcher*, *in-doubt-deny*),*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t*
and *simple-ruleset* (*lower-closure rs*)
— simple, normalized rules without unknowns
and ∀ *r* ∈ *set* (*lower-closure rs*). *normalized-nnf-match* (*get-match r*) ∧
normalized-src-ports (*get-match r*) ∧
normalized-dst-ports (*get-match r*) ∧
normalized-src-ips (*get-match r*) ∧
normalized-dst-ips (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*) ∧
¬ *has-disc is-Extra* (*get-match r*)
— no new primitives are introduced
and ∀ *a*. ¬ *disc* (*Src-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Dst-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Src*

```
a) ⟹ ∀ a. ¬ disc (Dst a) ⟹
      ∀ a. ¬ disc (IIface a) ∨ disc = is-Iiface ⟹ ∀ a. ¬ disc (OIface a) ∨ disc =
is-Oiface ⟹
      ∀ a. ¬ disc (Prot a) ⟹
       ∀ r ∈ set rs. ¬ has-disc disc (get-match r) ⟹
       ∀ r ∈ set (lower-closure rs). ¬ has-disc disc (get-match r)
 and ∀ a. ¬ disc (Src-Ports a) ⟹ ∀ a. ¬ disc (Dst-Ports a) ⟹ ∀ a. ¬ disc (Src
a) ⟹ ∀ a. ¬ disc (Dst a) ⟹
      ∀ a. ¬ disc (IIface a) ∨ disc = is-Iiface ⟹ ∀ a. ¬ disc (OIface a) ∨ disc =
is-Oiface ⟹
      (∀ a. ¬ disc (Prot a)) ∨ disc = is-Prot ∧
      (∀ r ∈ set rs. ¬ has-disc-negated is-Src-Ports False (get-match r) ∧
                  ¬ has-disc-negated is-Dst-Ports False (get-match r) ∧
                  ¬ has-disc is-MultiportPorts (get-match r)) ⟹
      ∀ r ∈ set rs. ¬ has-disc-negated disc False (get-match r) ⟹
      ∀ r ∈ set (lower-closure rs). ¬ has-disc-negated disc False (get-match r)
 proof −
  let ?rs1=optimize-matches-a lower-closure-matchexpr rs
  let ?rs2=transform-optimize-dnf-strict ?rs1
  let ?rs3=transform-normalize-primitives ?rs2
  let ?rs4=transform-optimize-dnf-strict ?rs3

  { fix m a
     have Rule m a ∈ set (lower-closure rs) ⟹
        (a = action.Accept ∨ a = action.Drop) ∧
         normalized-nnf-match m ∧
         normalized-src-ports m ∧
         normalized-dst-ports m ∧
         normalized-src-ips m ∧
         normalized-dst-ips m ∧
         ¬ has-disc is-MultiportPorts m ∧
          ¬ has-disc is-Extra m
     using simplers
     unfolding lower-closure-def
     apply(simp add: remdups-rev-set)
     apply(frule transform-remove-unknowns-lower(4))
     apply(drule transform-remove-unknowns-lower(2))
     thm transform-optimize-dnf-strict[OF - wf-in-doubt-deny]
    apply(frule(1) transform-optimize-dnf-strict-structure(2)[OF - wf-in-doubt-deny,
where disc=is-Extra])
      apply(thin-tac ∀ r∈ set (optimize-matches-a lower-closure-matchexpr rs). ¬
has-disc is-Extra (get-match r))
    apply(frule transform-optimize-dnf-strict-structure(3)[OF - wf-in-doubt-deny])
    apply(drule transform-optimize-dnf-strict-structure(1)[OF - wf-in-doubt-deny])
     thm transform-normalize-primitives[OF - wf-in-doubt-deny]
     apply(frule(1) transform-normalize-primitives(3)[OF - wf-in-doubt-deny, of
- is-Extra])
       apply(simp;fail)
      apply(simp;fail)
```

apply(*simp*;*fail*)
apply *blast*
apply(*thin-tac* ∀ *r*∈ *set ?rs2*. ¬ *has-disc is-Extra* (*get-match r*))
apply(*frule*(*1*) *transform-normalize-primitives*(*5*)[*OF - wf-in-doubt-deny*])
apply(*drule transform-normalize-primitives*(*2*)[*OF - wf-in-doubt-deny*], *simp*)
thm *transform-optimize-dnf-strict*[*OF - wf-in-doubt-deny*]
apply(*frule*(*1*) *transform-optimize-dnf-strict-structure*(*2*)[*OF - wf-in-doubt-deny*, **where** *disc*=*is-Extra*])
apply(*frule transform-optimize-dnf-strict-structure*(*2*)[*OF - wf-in-doubt-deny*, **where** *disc*=*is-MultiportPorts*])
apply *blast*
apply(*frule transform-optimize-dnf-strict-structure*(*3*)[*OF - wf-in-doubt-deny*])
apply(*frule transform-optimize-dnf-strict-structure*(*4*)[*OF - wf-in-doubt-deny*, *of -* (*is-Src-Ports*, *src-ports-sel*) (λ*ps. case ps of L4Ports - pts* ⇒ *length pts* ≤ *1*)])
apply(*simp add*: *normalized-src-ports-def2*; *fail*)
apply(*frule transform-optimize-dnf-strict-structure*(*4*)[*OF - wf-in-doubt-deny*, *of -* (*is-Dst-Ports*, *dst-ports-sel*) (λ*ps. case ps of L4Ports - pts* ⇒ *length pts* ≤ *1*)])
apply(*simp add*: *normalized-dst-ports-def2*; *fail*)
apply(*frule transform-optimize-dnf-strict-structure*(*4*)[*OF - wf-in-doubt-deny*, *of -* (*is-Src*, *src-sel*) *normalized-cidr-ip*])
apply(*simp add*: *normalized-src-ips-def2*; *fail*)
apply(*frule transform-optimize-dnf-strict-structure*(*4*)[*OF - wf-in-doubt-deny*, *of -* (*is-Dst*, *dst-sel*) *normalized-cidr-ip*])
apply(*simp add*: *normalized-dst-ips-def2*; *fail*)
apply(*thin-tac* ∀ *r*∈*set ?rs2*. *- r*)+
apply(*thin-tac* ∀ *r*∈*set ?rs3*. *- r*)+
apply(*drule transform-optimize-dnf-strict-structure*(*1*)[*OF - wf-in-doubt-deny*])
apply(*subgoal-tac* (*a* = *action.Accept* ∨ *a* = *action.Drop*))
prefer *2*
apply(*simp-all add*: *simple-ruleset-def*)
apply *fastforce*
apply(*simp add*: *normalized-src-ports-def2 normalized-dst-ports-def2 normalized-src-ips-def2 normalized-dst-ips-def2*)
apply(*intro conjI*)
by *fastforce*+
} **note** *1*=*this*

from *1* **show** *simple-ruleset* (*lower-closure rs*)
apply(*simp add*: *simple-ruleset-def*)
apply(*clarify*)
apply(*rename-tac r*)
apply(*case-tac r*)
apply(*simp*)
by *blast*

from *1* **show** ∀ *r* ∈ *set* (*lower-closure rs*). *normalized-nnf-match* (*get-match r*) ∧
normalized-src-ports (*get-match r*) ∧

normalized-dst-ports (*get-match r*) ∧
normalized-src-ips (*get-match r*) ∧
normalized-dst-ips (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*) ∧
¬ *has-disc is-Extra* (*get-match r*)
apply(*clarify*)
apply(*rename-tac r*)
apply(*case-tac r*)
apply(*simp*)
done

show ∀ *a*. ¬ *disc* (*Src-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Dst-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Src a*) ⟹ ∀ *a*. ¬ *disc* (*Dst a*) ⟹
∀ *a*. ¬ *disc* (*IIface a*) ∨ *disc* = *is-Iiface* ⟹ ∀ *a*. ¬ *disc* (*OIface a*) ∨ *disc* = *is-Oiface* ⟹
∀ *a*. ¬ *disc* (*Prot a*) ⟹
∀ *r* ∈ *set rs*. ¬ *has-disc disc* (*get-match r*) ⟹ ∀ *r* ∈ *set* (*lower-closure rs*). ¬ *has-disc disc* (*get-match r*)
using *simplers*
unfolding *lower-closure-def*
apply −
apply(*frule*(*1*) *transform-remove-unknowns-lower*(*3*)[**where** *disc*=*disc*])
apply(*drule transform-remove-unknowns-lower*(*2*))
apply(*frule*(*1*) *transform-optimize-dnf-strict-structure*(*2*)[*OF - wf-in-doubt-deny*, **where** *disc*=*disc*])
apply(*frule transform-optimize-dnf-strict-structure*(*3*)[*OF - wf-in-doubt-deny*])
apply(*drule transform-optimize-dnf-strict-structure*(*1*)[*OF - wf-in-doubt-deny*])
apply(*frule*(*1*) *transform-normalize-primitives*(*3*)[*OF - wf-in-doubt-deny*, *of - disc*])
apply(*simp*;*fail*)
apply *blast*
apply(*simp*;*fail*)
apply(*simp*;*fail*)
apply(*drule transform-normalize-primitives*(*2*)[*OF - wf-in-doubt-deny*], *simp*)
apply(*frule*(*1*) *transform-optimize-dnf-strict-structure*(*2*)[*OF - wf-in-doubt-deny*, **where** *disc*=*disc*])
apply(*simp add*: *remdups-rev-set*)
done

have *no-ports-1*:
¬ *has-disc-negated is-Src-Ports False* (*get-match m*) ∧
¬ *has-disc-negated is-Dst-Ports False* (*get-match m*) ∧
¬ *has-disc is-MultiportPorts* (*get-match m*)
if *no-ports*: ∀ *r*∈*set rs*.
¬ *has-disc-negated is-Src-Ports False* (*get-match r*) ∧
¬ *has-disc-negated is-Dst-Ports False* (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*)
and *m*: *m* ∈ *set* (*transform-optimize-dnf-strict* (*optimize-matches-a lower-closure-matchexpr*

rs))
for *m*
proof −
from *no-ports transform-remove-unknowns-lower*(*3*,*6*)[*OF simplers*] **have**
∀ *r*∈ *set* (*optimize-matches-a lower-closure-matchexpr rs*).
¬ *has-disc-negated is-Src-Ports False* (*get-match r*) ∧
¬ *has-disc-negated is-Dst-Ports False* (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*)
by *blast*
from *m this transform-optimize-dnf-strict-structure*(*2*,*5*)[*OF optimize-matches-a-simple-ruleset*[*OF simplers*] *wf-in-doubt-deny*, *of lower-closure-matchexpr*]
show *?thesis* **by** *blast*
qed

show∀ *a*. ¬ *disc* (*Src-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Dst-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Src a*) ⟹ ∀ *a*. ¬ *disc* (*Dst a*) ⟹
∀ *a*. ¬ *disc* (*IIface a*) ∨ *disc* = *is-Iiface* ⟹ ∀ *a*. ¬ *disc* (*OIface a*) ∨ *disc* = *is-Oiface* ⟹
(∀ *a*. ¬ *disc* (*Prot a*)) ∨ *disc* = *is-Prot* ∧
(∀ *r* ∈ *set rs*. ¬ *has-disc-negated is-Src-Ports False* (*get-match r*) ∧
¬ *has-disc-negated is-Dst-Ports False* (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*)) ⟹
∀ *r* ∈ *set rs*. ¬ *has-disc-negated disc False* (*get-match r*) ⟹
∀ *r* ∈ *set* (*lower-closure rs*). ¬ *has-disc-negated disc False* (*get-match r*)
using *simplers*
unfolding *lower-closure-def*
apply −
apply(*frule*(*1*) *transform-remove-unknowns-lower*(*6*)[**where** *disc*=*disc*])
apply(*drule transform-remove-unknowns-lower*(*2*))
apply(*frule*(*1*) *transform-optimize-dnf-strict-structure*(*5*)[*OF - wf-in-doubt-deny*, **where** *disc*=*disc*])
apply(*frule transform-optimize-dnf-strict-structure*(*3*)[*OF - wf-in-doubt-deny*])
apply(*drule transform-optimize-dnf-strict-structure*(*1*)[*OF - wf-in-doubt-deny*])
apply(*frule*(*1*) *transform-normalize-primitives*(*7*)[*OF - wf-in-doubt-deny*, *of - disc*])
apply(*simp*;*fail*)
apply *blast*
apply(*elim disjE*)
apply *blast*
apply(*simp*)
using *no-ports-1* **apply** *fast*
apply(*simp*;*fail*)
apply(*drule transform-normalize-primitives*(*2*)[*OF - wf-in-doubt-deny*], *simp*)
apply(*frule*(*1*) *transform-optimize-dnf-strict-structure*(*5*)[*OF - wf-in-doubt-deny*, **where** *disc*=*disc*])
apply(*simp add*: *remdups-rev-set*)
done

show (*common-matcher*, *in-doubt-deny*),*p*⊢ ⟨*lower-closure rs*, *s*⟩ ⇒$_\alpha$ *t* ⟷

(*common-matcher*, *in-doubt-deny*),*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t*
using *simplers*
unfolding *lower-closure-def*
apply −
apply(*frule transform-remove-unknowns-lower*(*1*)[**where** *p*=*p* **and** *s*=*s* **and** *t*=*t*])
apply(*drule transform-remove-unknowns-lower*(*2*))
apply(*frule transform-optimize-dnf-strict*[*OF* - *wf-in-doubt-deny*, **where** *p*=*p* **and** *s*=*s* **and** *t*=*t*])
apply(*frule transform-optimize-dnf-strict-structure*(*3*)[*OF* - *wf-in-doubt-deny*])
apply(*drule transform-optimize-dnf-strict-structure*(*1*)[*OF* - *wf-in-doubt-deny*])
apply(*frule*(*1*) *transform-normalize-primitives*(*1*)[*OF* - *wf-in-doubt-deny*, **where** *p*=*p* **and** *s*=*s* **and** *t*=*t*])
apply(*drule transform-normalize-primitives*(*2*)[*OF* - *wf-in-doubt-deny*], *simp*)
apply(*frule transform-optimize-dnf-strict*[*OF* - *wf-in-doubt-deny*, **where** *p*=*p* **and** *s*=*s* **and** *t*=*t*])
apply(*drule transform-optimize-dnf-strict-structure*(*1*)[*OF* - *wf-in-doubt-deny*])
apply(*simp*)
using *approximating-bigstep-fun-remdups-rev*
by (*simp add*: *approximating-bigstep-fun-remdups-rev approximating-semantics-iff-fun-good-ruleset remdups-rev-simplers simple-imp-good-ruleset*)
qed

definition *iface-try-rewrite*
:: (*iface* × (′*i*::*len word* × *nat*) *list*) *list*
⇒ ′*i prefix-routing option*
⇒ ′*i common-primitive rule list*
⇒ ′*i common-primitive rule list*
where
iface-try-rewrite ipassmt rtblo rs ≡
let o-rewrite = (*case rtblo of None* ⇒ *id* | *Some rtbl* ⇒
transform-optimize-dnf-strict ∘ *optimize-matches* (*oiface-rewrite* (*map-of-ipassmt* (*routing-ipassmt rtbl*)))) *in*
if ipassmt-sanity-disjoint (*map-of ipassmt*) ∧ *ipassmt-sanity-defined rs* (*map-of ipassmt*) *then*
optimize-matches (*iiface-rewrite* (*map-of-ipassmt ipassmt*)) (*o-rewrite rs*)
else
optimize-matches (*iiface-constrain* (*map-of-ipassmt ipassmt*)) (*o-rewrite rs*)

Where (*iface* × (′*i word* × *nat*) *list*) *list* is *map-of*′*i ipassignment*. The sanity checkers need to iterate over the interfaces, hence we don't pass a map but a list of tuples.

In `Transform.thy` there should be the final correctness theorem for *iface-try-rewrite*. Here are some structural properties.

lemma *iface-try-rewrite-simplers*: *simple-ruleset rs* ⟹ *simple-ruleset* (*iface-try-rewrite ipassmt rtblo rs*)
by(*simp add*: *iface-try-rewrite-def optimize-matches-simple-ruleset transform-optimize-dnf-strict-structure*(*1*)

```
- wf-in-doubt-allow
    ] Let-def split: option.splits)


lemma iiface-rewrite-preserves-nodisc:
  ∀ a. ¬ disc (Src a) ⟹ ¬ has-disc disc m ⟹ ¬ has-disc disc (iiface-rewrite
ipassmt m)
  proof(induction ipassmt m rule: iiface-rewrite.induct)
  case 2
   have ∀ a. ¬ disc (Src a) ⟹ ¬ disc (IIface ifce) ⟹ ¬ has-disc disc (ipassmt-iface-replace-srcip-mexpr
ipassmt ifce)
      for ifce ipassmt
      apply(simp add: ipassmt-iface-replace-srcip-mexpr-def split: option.split)
      apply(intro allI impI, rename-tac ips)
    apply(drule-tac X=Src and ls=map (uncurry IpAddrNetmask) ips in match-list-to-match-expr-not-has-dis
      apply(simp)
      done
    with 2 show ?case by simp
  qed(simp-all)

lemma iiface-constrain-preserves-nodisc:
  ∀ a. ¬ disc (Src a) ⟹ ¬ has-disc disc m ⟹ ¬ has-disc disc (iiface-constrain
ipassmt m)
  proof(induction ipassmt m rule: iiface-rewrite.induct)
  case 2
   have ∀ a. ¬ disc (Src a) ⟹ ¬ disc (IIface ifce) ⟹ ¬ has-disc disc (ipassmt-iface-constrain-srcip-mexpr
ipassmt ifce)
      for ifce ipassmt
      apply(simp add: ipassmt-iface-constrain-srcip-mexpr-def split: option.split)
      apply(intro allI impI, rename-tac ips)
    apply(drule-tac X=Src and ls=map (uncurry IpAddrNetmask) ips in match-list-to-match-expr-not-has-dis
      apply(simp)
      done
    with 2 show ?case by simp
  qed(simp-all)


lemma iface-try-rewrite-preserves-nodisc:
      simple-ruleset rs ⟹
      ∀ a. ¬ disc (Src a) ⟹ ∀ a. ¬ disc (Dst a) ⟹
      ∀ r∈ set rs. ¬ has-disc disc (get-match r) ⟹
       ∀ r∈ set (iface-try-rewrite ipassmt rtblo rs). ¬ has-disc disc (get-match r)
  apply(insert wf-in-doubt-deny)
  apply(simp add: iface-try-rewrite-def Let-def)
  apply(intro conjI impI optimize-matches-preserves)
  apply(case-tac[!] rtblo)
   apply(simp-all add: oiface-rewrite-preserves-nodisc iiface-rewrite-preserves-nodisc
iiface-constrain-preserves-nodisc)
   apply(rule iiface-rewrite-preserves-nodisc; assumption?)
```

apply(*rule transform-optimize-dnf-strict-structure*(*2*)[*THEN bspec*]; (*assumption*|*simp add*: *optimize-matches-simple-ruleset*; *fail*)*?*)
apply(*rule optimize-matches-preserves*)
apply(*rule oiface-rewrite-preserves-nodisc*; *simp*; *fail*)
apply(*rule iiface-constrain-preserves-nodisc*; *assumption?*)
apply(*rule transform-optimize-dnf-strict-structure*(*2*)[*THEN bspec*]; (*assumption*|*simp add*: *optimize-matches-simple-ruleset*; *fail*)*?*)
apply(*rule optimize-matches-preserves*)
apply(*rule oiface-rewrite-preserves-nodisc*; *simp*; *fail*)
done

theorem *iface-try-rewrite-no-rtbl*:
assumes *simplers*: *simple-ruleset rs*
and *normalized*: ∀ *r* ∈ *set rs*. *normalized-nnf-match* (*get-match r*)
and *wf-ipassmt1*: *ipassmt-sanity-nowildcards* (*map-of ipassmt*) **and** *wf-ipassmt2*: *distinct* (*map fst ipassmt*)
and *nospoofing*: ∃ *ips*. (*map-of ipassmt*) (*Iface* (*p-iiface p*)) = *Some ips* ∧ *p-src p* ∈ *ipcidr-union-set* (*set ips*)
shows (*common-matcher*, *α*),*p*⊢ ⟨*iface-try-rewrite ipassmt None rs*, *s*⟩ ⇒$_\alpha$ *t* ⟷ (*common-matcher*, *α*),*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t*
proof −
show (*common-matcher*, *α*),*p*⊢ ⟨*iface-try-rewrite ipassmt None rs*, *s*⟩ ⇒$_\alpha$ *t* ⟷ (*common-matcher*, *α*),*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t*
apply(*simp add*: *iface-try-rewrite-def Let-def comp-def*)
apply(*simp add*: *map-of-ipassmt-def wf-ipassmt1 wf-ipassmt2*)
apply(*intro conjI impI*)
apply(*elim conjE*)
using *iiface-rewrite*(*1*)[*OF simplers normalized wf-ipassmt1 - nospoofing*] **apply** *blast*
using *iiface-constrain*(*1*)[*OF simplers normalized wf-ipassmt1*, **where** *p* = *p*] *nospoofing* **apply** *force*
done
qed

lemma *optimize-matches-comp*:
assumes *mono*: ⋀*m*. *matcheq-matchNone m* ⟹ *matcheq-matchNone* (*g m*)
shows *optimize-matches* (*g* ∘ *f*) *rs* = *optimize-matches g* ((*optimize-matches f*) *rs*)
unfolding *optimize-matches-def*
proof(*induction rs*)
case (*Cons r rs*)
obtain *m a* **where** [*simp*]: *r* = *Rule m a* **by**(*cases r*)
show *?case*
proof(*cases matcheq-matchNone* (*f m*))
case *True*
hence *mn*: *matcheq-matchNone* (*g* (*f m*)) **by**(*fact mono*)
show *?thesis* **by**(*unfold comp-def* ; *simp add*: *mn Cons.IH*[*unfolded comp-def*])
next

case *False*
show *?thesis* **by**(*unfold comp-def*; *simp add*: *False Cons.IH*[*unfolded comp-def*])
qed
qed *simp*

context begin

private lemma *iiface-rewrite-monoNone*: *matcheq-matchNone m* ⟹ *matcheq-matchNone* (*iiface-rewrite ipassmt m*)
by(*induction m rule*: *matcheq-matchNone.induct*) *auto*
private lemma *iiface-constrain-monoNone*: *matcheq-matchNone m* ⟹ *matcheq-matchNone* (*iiface-constrain ipassmt m*)
by(*induction m rule*: *matcheq-matchNone.induct*) *auto*

private lemmas *optimize-matches-iiface-comp* = *optimize-matches-comp*[*OF iiface-rewrite-monoNone*]

optimize-matches-comp[*OF iiface-constrain-monoNone*]
end

theorem *iface-try-rewrite-rtbl*:
assumes *simplers*: *simple-ruleset rs*
and *normalized*: ∀ *r* ∈ *set rs*. *normalized-nnf-match* (*get-match r*)
and *wf-ipassmt*: *ipassmt-sanity-nowildcards* (*map-of ipassmt*) *distinct* (*map fst ipassmt*)
and *nospoofing*: ∃ *ips*. (*map-of ipassmt*) (*Iface* (*p-iiface p*)) = *Some ips* ∧ *p-src p* ∈ *ipcidr-union-set* (*set ips*)
and *routing-decided*: *output-iface* (*routing-table-semantics rtbl* (*p-dst p*)) = *p-oiface p*
and *correct-routing*: *correct-routing rtbl*
and *wf-ipassmt-o*: *ipassmt-sanity-nowildcards* (*map-of* (*routing-ipassmt rtbl*))
and *wf-match-tac*: *wf-unknown-match-tac* α
shows (*common-matcher*, α),*p*⊢ ⟨*iface-try-rewrite ipassmt* (*Some rtbl*) *rs*, *s*⟩ ⇒α *t* ⟷ (*common-matcher*, α),*p*⊢ ⟨*rs*, *s*⟩ ⇒α *t*
proof −
note *oiface-rewrite* = *oiface-rewrite*[*OF simplers normalized wf-ipassmt-o refl correct-routing routing-decided*]
let *?ors* = *optimize-matches* (*oiface-rewrite* (*map-of* (*routing-ipassmt rtbl*))) *rs*
let *?nrs* = *transform-optimize-dnf-strict ?ors*
have *osimplers*: *simple-ruleset ?ors* **using** *oiface-rewrite*(*2*) **.**
have *nsimplers*: *simple-ruleset ?nrs* **using** *transform-optimize-dnf-strict-structure*(*1*)[*OF osimplers wf-match-tac*] **.**
have *nnormalized*: ∀ *r* ∈ *set ?nrs*. *normalized-nnf-match* (*get-match r*) **using** *transform-optimize-dnf-strict-structure*(*3*)[*OF osimplers wf-match-tac*] **.**
note *nnf* = *transform-optimize-dnf-strict*[*OF osimplers wf-match-tac*]
have *nospoofing-alt*: ⋀*ips*. *map-of ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* ⟹ *p-src p* ∈ *ipcidr-union-set* (*set ips*) **using** *nospoofing* **by** *simp*
show (*common-matcher*, α),*p*⊢ ⟨*iface-try-rewrite ipassmt* (*Some rtbl*) *rs*, *s*⟩ ⇒α *t* ⟷ (*common-matcher*, α),*p*⊢ ⟨*rs*, *s*⟩ ⇒α *t*
apply(*simp add*: *iface-try-rewrite-def Let-def*)

apply(*simp add*: *map-of-ipassmt-def wf-ipassmt routing-ipassmt-distinct wf-ipassmt-o*)
apply(*intro conjI impI*; (*elim conjE*)*?*)
subgoal using *iiface-rewrite*(*1*)[*OF nsimplers nnormalized wf-ipassmt*(*1*) - *nospoofing*] *oiface-rewrite*(*1*) *nnf* **by** *simp*
subgoal using *iiface-constrain*(*1*)[*OF nsimplers nnormalized wf-ipassmt*(*1*), **where** *p* = *p*] *nospoofing-alt oiface-rewrite*(*1*) *nnf* **by** *simp*
done
qed

end
theory *Conntrack-State-Transform*
imports *Common-Primitive-Matcher*
../Semantics-Ternary/Semantics-Ternary
begin

The following function assumes that the packet is in a certain state.

fun *ctstate-assume-state* :: *ctstate* ⇒ *'i*::*len common-primitive match-expr* ⇒ *'i common-primitive match-expr* **where**
ctstate-assume-state s (*Match* (*CT-State x*)) = (*if s* ∈ *x then MatchAny else MatchNot MatchAny*) |
ctstate-assume-state s (*Match m*) = *Match m* |
ctstate-assume-state s (*MatchNot m*) = *MatchNot* (*ctstate-assume-state s m*) |
ctstate-assume-state - *MatchAny* = *MatchAny* |
ctstate-assume-state s (*MatchAnd m1 m2*) = *MatchAnd* (*ctstate-assume-state s m1*) (*ctstate-assume-state s m2*)

lemma *ctstate-assume-state*: *p-tag-ctstate p* = *s* ⟹
matches (*common-matcher*, *α*) (*ctstate-assume-state s m*) *a p* ⟷ *matches* (*common-matcher*, *α*) *m a p*
apply(*rule matches-iff-apply-f*)
by(*induction m rule*: *ctstate-assume-state.induct*) (*simp-all*)

definition *ctstate-assume-new* :: *'i*::*len common-primitive rule list* ⇒ *'i common-primitive rule list* **where**
ctstate-assume-new ≡ *optimize-matches* (*ctstate-assume-state CT-New*)

lemma *ctstate-assume-new-simple-ruleset*: *simple-ruleset rs* ⟹ *simple-ruleset* (*ctstate-assume-new rs*)
by (*simp add*: *ctstate-assume-new-def optimize-matches-simple-ruleset*)

Usually, the interesting part of a firewall is only about the rules for setting up connections. That means, we mostly only care about packets in state *CT-New*. Use the function *ctstate-assume-new* to remove all state matching and just care about the connection setup.

corollary *ctstate-assume-new*: *p-tag-ctstate p* = *CT-New* ⟹

approximating-bigstep-fun (*common-matcher*, α) *p* (*ctstate-assume-new rs*) *s* = *approximating-bigstep-fun* (*common-matcher*, α) *p rs s*
unfolding *ctstate-assume-new-def*
apply(*rule optimize-matches*)
apply(*simp add*: *ctstate-assume-state*)
done

If we assume the CT State is *CT-New*, we can also assume that the TCP SYN flag (*ipt-tcp-syn*) is set.

fun *ipt-tcp-flags-assume-flag* :: *ipt-tcp-flags* ⇒ *'i*::*len common-primitive match-expr* ⇒ *'i common-primitive match-expr* **where**
ipt-tcp-flags-assume-flag flg (*Match* (*L4-Flags x*)) = (*if ipt-tcp-flags-equal x flg then MatchAny else* (*case match-tcp-flags-conjunct-option x flg of None* ⇒ *MatchNot MatchAny* | *Some f3* ⇒ *Match* (*L4-Flags f3*))) |
ipt-tcp-flags-assume-flag flg (*Match m*) = *Match m* |
ipt-tcp-flags-assume-flag flg (*MatchNot m*) = *MatchNot* (*ipt-tcp-flags-assume-flag flg m*) |
ipt-tcp-flags-assume-flag - MatchAny = *MatchAny* |
ipt-tcp-flags-assume-flag flg (*MatchAnd m1 m2*) = *MatchAnd* (*ipt-tcp-flags-assume-flag flg m1*) (*ipt-tcp-flags-assume-flag flg m2*)

lemma *ipt-tcp-flags-assume-flag*: **assumes** *match-tcp-flags flg* (*p-tcp-flags p*)
shows *matches* (*common-matcher*, α) (*ipt-tcp-flags-assume-flag flg m*) *a p* ⟷ *matches* (*common-matcher*, α) *m a p*
proof(*rule matches-iff-apply-f*)
show *ternary-ternary-eval* (*map-match-tac common-matcher p* (*ipt-tcp-flags-assume-flag flg m*)) = *ternary-ternary-eval* (*map-match-tac common-matcher p m*)
using *assms* **proof**(*induction m rule*: *ipt-tcp-flags-assume-flag.induct*)
case (*1 flg x*)
thus *?case*
apply(*simp add*: *ipt-tcp-flags-equal del*: *match-tcp-flags.simps*)
apply(*cases match-tcp-flags-conjunct-option x flg*)
apply(*simp*)
using *match-tcp-flags-conjunct-option-None bool-to-ternary-simps*(*2*) **apply** *metis*
apply(*simp*)
apply(*drule-tac pkt*=(*p-tcp-flags p*) **in** *match-tcp-flags-conjunct-option-Some*)
by *simp*
qed(*simp-all del*: *match-tcp-flags.simps*)
qed

definition *ipt-tcp-flags-assume-syn* :: *'i*::*len common-primitive rule list* ⇒ *'i common-primitive rule list* **where**
ipt-tcp-flags-assume-syn ≡ *optimize-matches* (*ipt-tcp-flags-assume-flag ipt-tcp-syn*)

lemma *ipt-tcp-flags-assume-syn-simple-ruleset*: *simple-ruleset rs* ⟹ *simple-ruleset* (*ipt-tcp-flags-assume-syn rs*)
by (*simp add*: *ipt-tcp-flags-assume-syn-def optimize-matches-simple-ruleset*)

corollary *ipt-tcp-flags-assume-syn*: *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) $\Longrightarrow$
approximating-bigstep-fun (*common-matcher*, α) *p* (*ipt-tcp-flags-assume-syn rs*)
s = *approximating-bigstep-fun* (*common-matcher*, α) *p rs s*
unfolding *ipt-tcp-flags-assume-syn-def*
apply(*rule optimize-matches*)
apply(*simp add*: *ipt-tcp-flags-assume-flag*)
done

definition *packet-assume-new* :: *'i*::*len common-primitive rule list* $\Rightarrow$ *'i common-primitive rule list* **where**
packet-assume-new $\equiv$ *ctstate-assume-new* $\circ$ *ipt-tcp-flags-assume-syn*

lemma *packet-assume-new-simple-ruleset*: *simple-ruleset rs* $\Longrightarrow$ *simple-ruleset* (*packet-assume-new rs*)
by (*simp add*: *packet-assume-new-def ipt-tcp-flags-assume-syn-simple-ruleset ctstate-assume-new-simple-ruleset*)

corollary *packet-assume-new*: *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) $\Longrightarrow$ *p-tag-ctstate p* = *CT-New* $\Longrightarrow$
approximating-bigstep-fun (*common-matcher*, α) *p* (*packet-assume-new rs*) *s* = *approximating-bigstep-fun* (*common-matcher*, α) *p rs s*
unfolding *packet-assume-new-def*
by (*simp add*: *ctstate-assume-new ipt-tcp-flags-assume-syn*)

end
theory *Primitive-Abstract*
imports
Common-Primitive-toString
Transform
Conntrack-State-Transform
begin

40 Abstracting over Primitives

Abstract over certain primitives. The first parameter is a function *'i common-primitive negation-type* $\Rightarrow$ *bool* to select the primitives to be abstracted over. The *'i common-primitive* is wrapped in a *'i common-primitive negation-type* to let the function selectively abstract only over negated, non-negated, or both kinds of primitives. This functions requires a *normalized-nnf-match*.

fun *abstract-primitive*
:: (*'i*::*len common-primitive negation-type* ⇒ *bool*) ⇒ *'i common-primitive match-expr*
⇒ *'i common-primitive match-expr*
where
abstract-primitive - MatchAny = MatchAny |
abstract-primitive disc (*Match a*) =
(*if*
disc (*Pos a*)
then
Match (*Extra* (*common-primitive-toString ipaddr-generic-toString a*))
else
(*Match a*)) |
abstract-primitive disc (*MatchNot* (*Match a*)) =
(*if*
disc (*Neg a*)
then
Match (*Extra* (*''*! *''*@*common-primitive-toString ipaddr-generic-toString a*))
else
(*MatchNot* (*Match a*))) |
abstract-primitive disc (*MatchNot m*) = *MatchNot* (*abstract-primitive disc m*) |
abstract-primitive disc (*MatchAnd m1 m2*) = *MatchAnd* (*abstract-primitive disc m1*) (*abstract-primitive disc m2*)

For example, a simple firewall requires that no negated interfaces and protocols occur in the expression.

definition *abstract-for-simple-firewall* :: *'i*::*len common-primitive match-expr* ⇒ *'i common-primitive match-expr*
where *abstract-for-simple-firewall* ≡ *abstract-primitive* (λ*r*. *case r*
of Pos a ⇒ *is-CT-State a* ∨ *is-L4-Flags a*
| *Neg a* ⇒ *is-Iiface a* ∨ *is-Oiface a* ∨ *is-Prot a* ∨ *is-CT-State a* ∨ *is-L4-Flags a*)

lemma *abstract-primitive-preserves-normalized*:
normalized-src-ports m ⟹ *normalized-src-ports* (*abstract-primitive disc m*)
normalized-dst-ports m ⟹ *normalized-dst-ports* (*abstract-primitive disc m*)
normalized-src-ips m ⟹ *normalized-src-ips* (*abstract-primitive disc m*)
normalized-dst-ips m ⟹ *normalized-dst-ips* (*abstract-primitive disc m*)
normalized-nnf-match m ⟹ *normalized-nnf-match* (*abstract-primitive disc m*)
by(*induction disc m rule*: *abstract-primitive.induct*) (*simp-all*)
lemma *abstract-primitive-preserves-nodisc*:
¬ *has-disc disc' m* ⟹ (∀ *str*. ¬ *disc'* (*Extra str*)) ⟹ ¬ *has-disc disc'* (*abstract-primitive disc m*)
by(*induction disc m rule*: *abstract-primitive.induct*)(*simp-all*)
lemma *abstract-primitive-preserves-nodisc-nedgated*:
¬ *has-disc-negated disc' neg m* ⟹ (∀ *str*. ¬ *disc'* (*Extra str*)) ⟹ ¬ *has-disc-negated disc' neg* (*abstract-primitive disc m*)
by(*induction disc m arbitrary*: *neg rule*: *abstract-primitive.induct*) *simp+*

lemma *abstract-primitive-nodisc*:
∀ *x*. *disc′ x* ⟶ *disc* (*Pos x*) ∧ *disc* (*Neg x*) ⟹ (∀ *str*. ¬ *disc′* (*Extra str*)) ⟹ ¬ *has-disc disc′* (*abstract-primitive disc m*)
by(*induction disc m rule*: *abstract-primitive.induct*) *auto*

lemma *abstract-primitive-preserves-not-has-disc-negated*:
∀ *a*. ¬ *disc* (*Extra a*)⟹ ¬ *has-disc-negated disc neg m* ⟹ ¬ *has-disc-negated disc neg* (*abstract-primitive sel-f m*)
by(*induction sel-f m arbitrary*: *neg rule*: *abstract-primitive.induct*) *simp+*

lemma *abstract-for-simple-firewall-preserves-nodisc-negated*:
∀ *a*. ¬ *disc* (*Extra a*)⟹ ¬ *has-disc-negated disc False m* ⟹ ¬ *has-disc-negated disc False* (*abstract-for-simple-firewall m*)
unfolding *abstract-for-simple-firewall-def*
using *abstract-primitive-preserves-nodisc-nedgated* **by** *blast*

The function *ctstate-assume-state* can be used to fix a state and hence remove all state matches from the ruleset. It is therefore advisable to create a simple firewall for a fixed state, e.g. with *ctstate-assume-new* before calling to *abstract-for-simple-firewall*.

lemma *not-hasdisc-ctstate-assume-state*: ¬ *has-disc is-CT-State* (*ctstate-assume-state s m*)
by(*induction m rule*: *ctstate-assume-state.induct*) (*simp-all*)

lemma *abstract-for-simple-firewall-hasdisc*: **fixes** *m* :: *′i*::*len common-primitive match-expr*
shows ¬ *has-disc is-CT-State* (*abstract-for-simple-firewall m*)
and ¬ *has-disc is-L4-Flags* (*abstract-for-simple-firewall m*)
unfolding *abstract-for-simple-firewall-def*
apply(*induction* (λ*r*:: *′i common-primitive negation-type*. *case r of Pos a* ⇒ *is-CT-State a* | *Neg a* ⇒ *is-Iiface a* ∨ *is-Oiface a* ∨ *is-Prot a* ∨ *is-CT-State a*) *m rule*: *abstract-primitive.induct*)
apply(*simp-all*)
done

lemma *abstract-for-simple-firewall-negated-ifaces-prots*: **fixes** *m* :: *′i*::*len common-primitive match-expr*
shows *normalized-nnf-match m* ⟹ ¬ *has-disc-negated* (λ*a*. *is-Iiface a* ∨ *is-Oiface a*) *False* (*abstract-for-simple-firewall m*)
and *normalized-nnf-match m* ⟹ ¬ *has-disc-negated is-Prot False* (*abstract-for-simple-firewall m*)
unfolding *abstract-for-simple-firewall-def*
apply(*induction* (λ*r*:: *′i common-primitive negation-type*. *case r of Pos a* ⇒ *is-CT-State a* | *Neg a* ⇒ *is-Iiface a* ∨ *is-Oiface a* ∨ *is-Prot a* ∨ *is-CT-State a*) *m rule*: *abstract-primitive.induct*)
apply(*simp-all*)
done

context
begin
private lemma *abstract-primitive-in-doubt-allow-Allow*:
primitive-matcher-generic $\beta \Longrightarrow$ *normalized-nnf-match* $m \Longrightarrow$
matches (β, *in-doubt-allow*) m *action.Accept* $p \Longrightarrow$
matches (β, *in-doubt-allow*) (*abstract-primitive disc* m) *action.Accept* p
by(*induction disc m rule*: *abstract-primitive.induct*)
(*simp-all add*: *bunch-of-lemmata-about-matches*(*1*) *primitive-matcher-generic.Extra-single*)

private lemma *abstract-primitive-in-doubt-allow-Allow2*:
primitive-matcher-generic $\beta \Longrightarrow$ *normalized-nnf-match* $m \Longrightarrow$
$\neg$ *matches* (β, *in-doubt-allow*) m *action.Drop* $p \Longrightarrow$
$\neg$ *matches* (β, *in-doubt-allow*) (*abstract-primitive disc* m) *action.Drop* p
proof(*induction disc m rule*: *abstract-primitive.induct*)
case(*5 m1 m2*) **thus** *?case* **by** (*auto simp add*: *bunch-of-lemmata-about-matches*(*1*))
qed(*simp-all add*: *bunch-of-lemmata-about-matches*(*1*) *primitive-matcher-generic.Extra-single*)

private lemma *abstract-primitive-in-doubt-allow-Deny*:
primitive-matcher-generic $\beta \Longrightarrow$ *normalized-nnf-match* $m \Longrightarrow$
matches (β, *in-doubt-allow*) (*abstract-primitive disc* m) *action.Drop* $p \Longrightarrow$
matches (β, *in-doubt-allow*) m *action.Drop* p
apply(*induction disc m rule*: *abstract-primitive.induct*)
apply (*simp-all add*: *bunch-of-lemmata-about-matches*(*1*))
apply(*auto simp add*: *primitive-matcher-generic.Extra-single primitive-matcher-generic.Extra-single-not split*: *if-split-asm*)
done

private lemma *abstract-primitive-in-doubt-allow-Deny2*:
primitive-matcher-generic $\beta \Longrightarrow$ *normalized-nnf-match* $m \Longrightarrow$
$\neg$ *matches* (β, *in-doubt-allow*) (*abstract-primitive disc* m) *action.Accept* $p \Longrightarrow$
$\neg$ *matches* (β, *in-doubt-allow*) m *action.Accept* p
apply(*induction disc m rule*: *abstract-primitive.induct*)
apply (*simp-all add*: *bunch-of-lemmata-about-matches*(*1*))
apply(*auto simp add*: *primitive-matcher-generic.Extra-single primitive-matcher-generic.Extra-single-not split*: *if-split-asm*)
done

theorem *abstract-primitive-in-doubt-allow-generic*:
fixes β::(*'i*::*len common-primitive*, (*'i*, *'a*) *tagged-packet-scheme*) *exact-match-tac*
assumes *generic*: *primitive-matcher-generic* β
and n: $\forall\ r \in$ *set rs*. *normalized-nnf-match* (*get-match r*)
and *simple*: *simple-ruleset rs*
defines $\gamma \equiv$ (β, *in-doubt-allow*) **and** *abstract disc* $\equiv$ *optimize-matches* (*abstract-primitive disc*)
shows {*p*. γ,*p*$\vdash$ $\langle$*abstract disc rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalDeny*} $\subseteq$ {*p*. γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalDeny*}
(**is** *?deny*)
and {*p*. γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalAllow*} $\subseteq$ {*p*. γ,*p*$\vdash$ $\langle$*abstract*

```
disc rs, Undecided⟩ ⇒α Decision FinalAllow}
            (is ?allow)
  proof −
    from simple have good-ruleset rs using simple-imp-good-ruleset by fast
    from optimize-matches-simple-ruleset simple simple-imp-good-ruleset have
     good: good-ruleset (optimize-matches (abstract-primitive disc) rs) by fast

  let ?γ=(β, in-doubt-allow) :: ('i::len common-primitive, ('i, 'a) tagged-packet-scheme)
match-tac
      — type signature is needed, otherwise in-doubt-allow would be for arbitrary
packet

    have abstract-primitive-in-doubt-allow-help1:
      approximating-bigstep-fun γ p (optimize-matches (abstract-primitive disc)
rs) Undecided = Decision FinalAllow
      if prem: approximating-bigstep-fun γ p rs Undecided = Decision FinalAllow
for p
     proof −
         from simple have wf-ruleset γ p rs using good-imp-wf-ruleset sim-
ple-imp-good-ruleset by fast
       from this simple prem n show ?thesis
        unfolding γ-def
      proof(induction ?γ p rs Undecided rule: approximating-bigstep-fun-induct-wf)
        case (MatchAccept p m a rs)
         from MatchAccept.prems
          abstract-primitive-in-doubt-allow-Allow[OF generic] MatchAccept.hyps
have
           matches ?γ (abstract-primitive disc m) action.Accept p by simp
         thus ?case
         apply(simp add: MatchAccept.hyps(2))
         using optimize-matches-matches-fst by fastforce
        next
        case (Nomatch p m a rs) thus ?case
         proof(cases matches ?γ (abstract-primitive disc m) a p)
          case False with Nomatch show ?thesis
           apply(simp add: optimize-matches-def)
           using simple-ruleset-tail by blast
          next
          case True
          from Nomatch.prems(1) have a = action.Accept ∨ a = action.Drop
by(simp add: simple-ruleset-def)
         from Nomatch.hyps(1) Nomatch.prems(3) abstract-primitive-in-doubt-allow-Allow2[OF
generic] have
              a = action.Drop ⟹ ¬ matches ?γ (abstract-primitive disc m)
action.Drop p by simp
              with True ‹a = action.Accept ∨ a = action.Drop› have a =
action.Accept by blast
            with True show ?thesis
             using optimize-matches-matches-fst by fastforce
```

```
        qed
      qed(simp-all add: simple-ruleset-def)
  qed

  have abstract-primitive-in-doubt-allow-help2:
    approximating-bigstep-fun γ p rs Undecided = Decision FinalDeny
   if prem: approximating-bigstep-fun γ p (optimize-matches (abstract-primitive
disc) rs) Undecided = Decision FinalDeny
   for p
   proof −
      from simple have wf-ruleset γ p rs using good-imp-wf-ruleset sim-
ple-imp-good-ruleset by fast
     from this simple prem n show ?thesis
      unfolding γ-def
    proof(induction ?γ p rs Undecided rule: approximating-bigstep-fun-induct-wf)
      case Empty thus ?case by(simp add: optimize-matches-def)
      next
      case (MatchAccept p m a rs)
        from MatchAccept.prems abstract-primitive-in-doubt-allow-Allow[OF
generic] MatchAccept.hyps have
          1: matches ?γ (abstract-primitive disc m) action.Accept p by simp
        with MatchAccept have approximating-bigstep-fun ?γ p
          (Rule (abstract-primitive disc m) action.Accept # (optimize-matches
(abstract-primitive disc) rs)) Undecided = Decision FinalDeny
          using optimize-matches-matches-fst by metis
        with 1 have False by(simp)
        thus ?case ..
      next
      case (Nomatch p m a rs) thus ?case
        proof(cases matches ?γ (abstract-primitive disc m) a p)
          case False
            with Nomatch.prems(2) have approximating-bigstep-fun ?γ p
(optimize-matches (abstract-primitive disc) rs) Undecided = Decision FinalDeny
            by(simp add: optimize-matches-def split: if-split-asm)
          with Nomatch have IH: approximating-bigstep-fun ?γ p rs Undecided
= Decision FinalDeny
            using simple-ruleset-tail by auto
          with Nomatch(1) show ?thesis by simp
          next
          case True
            from Nomatch.prems(2) True have 1: approximating-bigstep-fun
?γ p
                (Rule (abstract-primitive disc m) a # (optimize-matches
(abstract-primitive disc) rs)) Undecided = Decision FinalDeny
              using optimize-matches-matches-fst by metis

          from Nomatch.prems(1) have a = action.Accept ∨ a = action.Drop
by(simp add: simple-ruleset-def)
        from Nomatch.hyps(1) Nomatch.prems(3) abstract-primitive-in-doubt-allow-Allow2[OF
```

generic] **have**
a = action.Drop ⟹ ¬ matches ?γ (abstract-primitive disc m) action.Drop p **by** *simp*
with *True ‹a = action.Accept ∨ a = action.Drop›* **have** *a = action.Accept* **by** *blast*
with *1 True* **have** *False* **by** *simp*
thus *?thesis* **..**
qed
qed(*simp-all add*: *simple-ruleset-def*)
qed

from *good approximating-semantics-iff-fun-good-ruleset abstract-primitive-in-doubt-allow-help1 ‹good-ruleset rs›* **show** *?allow*
unfolding *abstract-def* **by** *fast*
from *good approximating-semantics-iff-fun-good-ruleset abstract-primitive-in-doubt-allow-help2 ‹good-ruleset rs› γ-def* **show** *?deny*
unfolding *abstract-def* **by** *fast*
qed
corollary *abstract-primitive-in-doubt-allow*:
assumes ∀ *r* ∈ *set rs. normalized-nnf-match* (*get-match r*) **and** *simple-ruleset rs*
defines γ ≡ (*common-matcher*, *in-doubt-allow*) **and** *abstract disc* ≡ *optimize-matches* (*abstract-primitive disc*)
shows {*p*. γ,*p*⊢ ⟨*abstract disc rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalDeny*} ⊆ {*p*. γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalDeny*}
and {*p*. γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow*} ⊆ {*p*. γ,*p*⊢ ⟨*abstract disc rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow*}
unfolding γ-*def abstract-def*
using *assms abstract-primitive-in-doubt-allow-generic*[*OF primitive-matcher-generic-common-matcher*]
by *blast*+
end

context
begin
private lemma *abstract-primitive-in-doubt-deny-Deny*:
primitive-matcher-generic β ⟹ normalized-nnf-match m ⟹
matches (*β*, *in-doubt-deny*) *m action.Drop p ⟹*
matches (*β*, *in-doubt-deny*) (*abstract-primitive disc m*) *action.Drop p*
by(*induction disc m rule*: *abstract-primitive.induct*)
(*simp-all add*: *bunch-of-lemmata-about-matches*(*1*) *primitive-matcher-generic.Extra-single*)

private lemma *abstract-primitive-in-doubt-deny-Deny2*:
primitive-matcher-generic β ⟹ normalized-nnf-match m ⟹
¬ matches (*β*, *in-doubt-deny*) *m action.Accept p ⟹*
¬ matches (*β*, *in-doubt-deny*) (*abstract-primitive disc m*) *action.Accept p*
proof(*induction disc m rule*: *abstract-primitive.induct*)
case(*5 m1 m2*) **thus** *?case* **by** (*auto simp add*: *bunch-of-lemmata-about-matches*(*1*))
qed(*simp-all add*: *bunch-of-lemmata-about-matches*(*1*) *primitive-matcher-generic.Extra-single*)

private lemma *abstract-primitive-in-doubt-deny-Allow*:
primitive-matcher-generic $\beta \Longrightarrow$ *normalized-nnf-match* $m \Longrightarrow$
matches (β, *in-doubt-deny*) (*abstract-primitive disc m*) *action.Accept p* $\Longrightarrow$
matches (β, *in-doubt-deny*) *m action.Accept p*
apply(*induction disc m rule*: *abstract-primitive.induct*)
apply (*simp-all add*: *bunch-of-lemmata-about-matches*(*1*))
apply(*auto simp add*: *primitive-matcher-generic.Extra-single primitive-matcher-generic.Extra-single-not split*: *if-split-asm*)
done

private lemma *abstract-primitive-in-doubt-deny-Allow2*:
primitive-matcher-generic $\beta \Longrightarrow$ *normalized-nnf-match* $m \Longrightarrow$
$\neg$ *matches* (β, *in-doubt-deny*) (*abstract-primitive disc m*) *action.Drop p* $\Longrightarrow$
$\neg$ *matches* (β, *in-doubt-deny*) *m action.Drop p*
apply(*induction disc m rule*: *abstract-primitive.induct*)
apply (*simp-all add*: *bunch-of-lemmata-about-matches*(*1*))
apply(*auto simp add*: *primitive-matcher-generic.Extra-single primitive-matcher-generic.Extra-single-not split*: *if-split-asm*)
done

theorem *abstract-primitive-in-doubt-deny-generic*:
fixes β::('*i*::*len common-primitive*, ('*i*, '*a*) *tagged-packet-scheme*) *exact-match-tac*
assumes *generic*: *primitive-matcher-generic* β
and *n*: $\forall$ *r* $\in$ *set rs*. *normalized-nnf-match* (*get-match r*)
and *simple*: *simple-ruleset rs*
defines $\gamma \equiv$ (β, *in-doubt-deny*) **and** *abstract disc* $\equiv$ *optimize-matches* (*abstract-primitive disc*)
shows {*p*. γ,*p*$\vdash$ $\langle$*abstract disc rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalAllow*} $\subseteq$ {*p*. γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalAllow*}
(**is** *?allow*)
and {*p*. γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalDeny*} $\subseteq$ {*p*. γ,*p*$\vdash$ $\langle$*abstract disc rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalDeny*}
(**is** *?deny*)
proof −
from *simple* **have** *good-ruleset rs* **using** *simple-imp-good-ruleset* **by** *fast*
from *optimize-matches-simple-ruleset simple simple-imp-good-ruleset* **have**
good: *good-ruleset* (*optimize-matches* (*abstract-primitive disc*) *rs*) **by** *fast*

let *?*γ=(β, *in-doubt-deny*) :: ('*i*::*len common-primitive*, ('*i*, '*a*) *tagged-packet-scheme*) *match-tac*
— type signature is needed, otherwise *in-doubt-allow* would be for arbitrary packet

have *abstract-primitive-in-doubt-deny-help1*:
approximating-bigstep-fun γ *p* (*optimize-matches* (*abstract-primitive disc*) *rs*) *Undecided* = *Decision FinalDeny*
if *prem*: *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalDeny* **for** *p*

```
    proof −
        from simple have wf-ruleset γ p rs using good-imp-wf-ruleset simple-imp-good-ruleset by fast
      from this simple prem n show ?thesis
        unfolding γ-def
      proof(induction ?γ p rs Undecided rule: approximating-bigstep-fun-induct-wf)
        case (MatchDrop p m a rs)
            from MatchDrop.prems abstract-primitive-in-doubt-deny-Deny[OF generic] MatchDrop.hyps have
              matches ?γ (abstract-primitive disc m) action.Drop p by simp
          thus ?case
          apply(simp add: MatchDrop.hyps(2))
          using optimize-matches-matches-fst by fastforce
        next
        case (Nomatch p m a rs) thus ?case
          proof(cases matches ?γ (abstract-primitive disc m) a p)
            case False with Nomatch show ?thesis
              apply(simp add: optimize-matches-def)
              using simple-ruleset-tail by blast
            next
            case True
             from Nomatch.prems(1) have a = action.Accept ∨ a = action.Drop
by(simp add: simple-ruleset-def)
           from Nomatch.hyps(1) Nomatch.prems(3) abstract-primitive-in-doubt-deny-Deny2[OF generic] have
                a = action.Accept ⟹ ¬ matches ?γ (abstract-primitive disc m) action.Accept p by(simp)
                  with True ‹a = action.Accept ∨ a = action.Drop› have a = action.Drop by blast
                with True show ?thesis using optimize-matches-matches-fst by fastforce
            qed
        qed(simp-all add: simple-ruleset-def)
    qed

    have abstract-primitive-in-doubt-deny-help2:
      approximating-bigstep-fun γ p rs Undecided = Decision FinalAllow
     if prem: approximating-bigstep-fun γ p (optimize-matches (abstract-primitive disc) rs) Undecided = Decision FinalAllow
      for p
      proof −
          from simple have wf-ruleset ?γ p rs using good-imp-wf-ruleset simple-imp-good-ruleset by fast
        from this simple prem n show ?thesis
          unfolding γ-def
        proof(induction ?γ p rs Undecided rule: approximating-bigstep-fun-induct-wf)
          case Empty thus ?case by(simp add: optimize-matches-def)
          next
          case (MatchAccept p m a rs) thus ?case by auto
```

```
          next
          case (MatchDrop p m a rs)
               from MatchDrop.prems abstract-primitive-in-doubt-deny-Deny[OF
generic] MatchDrop.hyps have
              1: matches ?γ (abstract-primitive disc m) action.Drop p by simp
            from MatchDrop have approximating-bigstep-fun ?γ p
              (Rule (abstract-primitive disc m) action.Drop # (optimize-matches
(abstract-primitive disc) rs)) Undecided = Decision FinalAllow
            using optimize-matches-matches-fst 1 by fastforce
            with 1 have False by(simp)
            thus ?case ..
          next
          case (Nomatch p m a rs) thus ?case
            proof(cases matches ?γ (abstract-primitive disc m) a p)
              case False
                   with Nomatch.prems(2) have approximating-bigstep-fun ?γ p
(optimize-matches (abstract-primitive disc) rs) Undecided = Decision FinalAllow
                by(simp add: optimize-matches-def split: if-split-asm)
              with Nomatch have IH: approximating-bigstep-fun ?γ p rs Undecided
= Decision FinalAllow
                using simple-ruleset-tail by auto
              with Nomatch(1) show ?thesis by simp
              next
              case True
                 from Nomatch.prems(2) True have 1: approximating-bigstep-fun
?γ p
                           (Rule (abstract-primitive disc m) a # (optimize-matches
(abstract-primitive disc) rs)) Undecided = Decision FinalAllow
                  using optimize-matches-matches-fst by metis
               from Nomatch.prems(1) have a = action.Accept ∨ a = action.Drop
by(simp add: simple-ruleset-def)
             from Nomatch.hyps(1) Nomatch.prems(3) abstract-primitive-in-doubt-deny-Deny2[OF
generic] have
                   a = action.Accept ⟹ ¬ matches ?γ (abstract-primitive disc m)
action.Accept p by simp
                  with True ‹a = action.Accept ∨ a = action.Drop› have a =
action.Drop by blast
                with 1 True have False by force
                thus ?thesis ..
              qed
          qed(simp-all add: simple-ruleset-def)
      qed

    from good approximating-semantics-iff-fun-good-ruleset abstract-primitive-in-doubt-deny-help1
‹good-ruleset rs› show ?deny
        unfolding abstract-def by fast
    from good approximating-semantics-iff-fun-good-ruleset abstract-primitive-in-doubt-deny-help2
‹good-ruleset rs› show ?allow
        unfolding abstract-def by fast
```

qed
end

end

41 Iptables to Simple Firewall and Vice Versa

theory *SimpleFw-Compliance*
imports *Simple-Firewall.SimpleFw-Semantics*
../Primitive-Matchers/Transform
../Primitive-Matchers/Primitive-Abstract
begin

41.1 Simple Match to MatchExpr

fun *simple-match-to-ipportiface-match* :: *'i::len simple-match* ⇒ *'i common-primitive match-expr* **where**
simple-match-to-ipportiface-match (⦇*iiface=iif*, *oiface=oif*, *src=sip*, *dst=dip*, *proto=p*, *sports=sps*, *dports=dps* ⦈) =
MatchAnd (*Match* (*IIface iif*)) (*MatchAnd* (*Match* (*OIface oif*))
(*MatchAnd* (*Match* (*Src* (*uncurry IpAddrNetmask sip*)))
(*MatchAnd* (*Match* (*Dst* (*uncurry IpAddrNetmask dip*)))
(*case p of ProtoAny* ⇒ *MatchAny*
| *Proto prim-p* ⇒
(*MatchAnd* (*Match* (*Prot p*))
(*MatchAnd* (*Match* (*Src-Ports* (*L4Ports prim-p* [*sps*])))
(*Match* (*Dst-Ports* (*L4Ports prim-p* [*dps*])))
))
))))

lemma *ports-to-set-singleton-simple-match-port*: *p* ∈ *ports-to-set* [*a*] ⟷ *simple-match-port a p*
by(*cases a*, *simp*)

theorem *simple-match-to-ipportiface-match-correct*:
assumes *valid*: *simple-match-valid sm*
shows *matches* (*common-matcher*, *α*) (*simple-match-to-ipportiface-match sm*) *a p* ⟷ *simple-matches sm p*
proof −
obtain *iif oif sip dip pro sps dps* **where**
sm: *sm* = ⦇*iiface* = *iif*, *oiface* = *oif*, *src* = *sip*, *dst* = *dip*, *proto* = *pro*, *sports* = *sps*, *dports* = *dps*⦈ **by** (*cases sm*)
{ fix *ip*
have *p-src p* ∈ *ipt-iprange-to-set* (*uncurry IpAddrNetmask ip*) ⟷ *simple-match-ip ip* (*p-src p*)
and *p-dst p* ∈ *ipt-iprange-to-set* (*uncurry IpAddrNetmask ip*) ⟷ *simple-match-ip ip* (*p-dst p*)

by(*simp split*: *uncurry-split*)+
} note *simple-match-ips=this*
{ fix *ps*
have *p-sport p* ∈ *ports-to-set* [*ps*] ⟷ *simple-match-port ps* (*p-sport p*)
and *p-dport p* ∈ *ports-to-set* [*ps*] ⟷ *simple-match-port ps* (*p-dport p*)
apply(*case-tac* [!] *ps*)
by(*simp-all*)
} note *simple-match-ports=this*
from *valid sm* **have** *valid'*:*pro* = *ProtoAny* ⟹ *simple-match-port sps* (*p-sport p*) ∧ *simple-match-port dps* (*p-dport p*)
apply(*simp add*: *simple-match-valid-def*)
by *blast*
show *?thesis* **unfolding** *sm*
apply(*cases pro*)
subgoal
apply(*simp add*: *bunch-of-lemmata-about-matches simple-matches.simps*)
apply(*simp add*: *match-raw-bool ternary-to-bool-bool-to-ternary simple-match-ips simple-match-ports simple-matches.simps*)
using *valid'* **by** *simp*
apply(*simp add*: *bunch-of-lemmata-about-matches simple-matches.simps*)
apply(*simp add*: *match-raw-bool ternary-to-bool-bool-to-ternary simple-match-ips simple-match-ports simple-matches.simps*)
apply *fast*
done
qed

41.2 MatchExpr to Simple Match

fun *common-primitive-match-to-simple-match* :: *'i*::*len common-primitive match-expr* ⇒ *'i simple-match option* **where**
common-primitive-match-to-simple-match MatchAny = *Some* (*simple-match-any*) |
common-primitive-match-to-simple-match (*MatchNot MatchAny*) = *None* |
common-primitive-match-to-simple-match (*Match* (*IIface iif*)) = *Some* (*simple-match-any*⦇ *iiface* := *iif* ⦈) |
common-primitive-match-to-simple-match (*Match* (*OIface oif*)) = *Some* (*simple-match-any*⦇ *oiface* := *oif* ⦈) |
common-primitive-match-to-simple-match (*Match* (*Src* (*IpAddrNetmask pre len*))) = *Some* (*simple-match-any*⦇ *src* := (*pre*, *len*) ⦈) |
common-primitive-match-to-simple-match (*Match* (*Dst* (*IpAddrNetmask pre len*))) = *Some* (*simple-match-any*⦇ *dst* := (*pre*, *len*) ⦈) |
common-primitive-match-to-simple-match (*Match* (*Prot p*)) = *Some* (*simple-match-any*⦇ *proto* := *p* ⦈) |
common-primitive-match-to-simple-match (*Match* (*Src-Ports* (*L4Ports p* []))) = *None* |
common-primitive-match-to-simple-match (*Match* (*Src-Ports* (*L4Ports p* [(*s*,*e*)]))) = *Some* (*simple-match-any*⦇ *proto* := *Proto p*, *sports* := (*s*,*e*) ⦈) |
common-primitive-match-to-simple-match (*Match* (*Dst-Ports* (*L4Ports p* []))) = *None* |

common-primitive-match-to-simple-match (*Match* (*Dst-Ports* (*L4Ports p* [(*s*,*e*)]))) = *Some* (*simple-match-any*⦇ *proto* := *Proto p*, *dports* := (*s*,*e*) ⦈) |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*Prot ProtoAny*))) = *None* |
common-primitive-match-to-simple-match (*MatchAnd m1 m2*) = (*case* (*common-primitive-match-to-simple-m m1*, *common-primitive-match-to-simple-match m2*) *of*
(*None*, -) ⇒ *None*
| (-, *None*) ⇒ *None*
| (*Some m1'*, *Some m2'*) ⇒ *simple-match-and m1' m2'*) |
— undefined cases, normalize before!
common-primitive-match-to-simple-match (*Match* (*Src* (*IpAddr* -))) = *undefined* |
common-primitive-match-to-simple-match (*Match* (*Src* (*IpAddrRange* - -))) = *undefined* |
common-primitive-match-to-simple-match (*Match* (*Dst* (*IpAddr* -))) = *undefined* |
common-primitive-match-to-simple-match (*Match* (*Dst* (*IpAddrRange* - -))) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*Prot* -))) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*IIface* -))) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*OIface* -))) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*Src* -))) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*Dst* -))) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*MatchAnd* - -)) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*MatchNot* -)) = *undefined* |
common-primitive-match-to-simple-match (*Match* (*Src-Ports* -)) = *undefined* |
common-primitive-match-to-simple-match (*Match* (*Dst-Ports* -)) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*Src-Ports* -))) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*Dst-Ports* -))) = *undefined* |
common-primitive-match-to-simple-match (*Match* (*CT-State* -)) = *undefined* |
common-primitive-match-to-simple-match (*Match* (*L4-Flags* -)) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*L4-Flags* -))) = *undefined* |
common-primitive-match-to-simple-match (*Match* (*Extra* -)) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*Extra* -))) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*CT-State* -))) = *undefined*

41.2.1 Normalizing Interfaces

As for now, negated interfaces are simply not allowed

definition *normalized-ifaces* :: *'i::len common-primitive match-expr* ⇒ *bool* **where**
normalized-ifaces m ≡ ¬ *has-disc-negated* (λ*a*. *is-Iiface a* ∨ *is-Oiface a*) *False m*

41.2.2 Normalizing Protocols

As for now, negated protocols are simply not allowed

definition *normalized-protocols* :: *'i::len common-primitive match-expr* ⇒ *bool* **where**
normalized-protocols m ≡ ¬ *has-disc-negated is-Prot False m*

lemma *match-iface-simple-match-any-simps*:
match-iface (*iiface simple-match-any*) (*p-iiface p*)
match-iface (*oiface simple-match-any*) (*p-oiface p*)
simple-match-ip (*src simple-match-any*) (*p-src p*)
simple-match-ip (*dst simple-match-any*) (*p-dst p*)
match-proto (*proto simple-match-any*) (*p-proto p*)
simple-match-port (*sports simple-match-any*) (*p-sport p*)
simple-match-port (*dports simple-match-any*) (*p-dport p*)
apply (*simp-all add*: *simple-match-any-def match-ifaceAny ipset-from-cidr-0*)
apply (*subgoal-tac* [!] (*65535*::*16 word*) = − *1*)
apply (*simp-all only*:)
apply *simp-all*
done

theorem *common-primitive-match-to-simple-match*:
assumes *normalized-src-ports m*
and *normalized-dst-ports m*
and *normalized-src-ips m*
and *normalized-dst-ips m*
and *normalized-ifaces m*
and *normalized-protocols m*
and ¬ *has-disc is-L4-Flags m*
and ¬ *has-disc is-CT-State m*
and ¬ *has-disc is-MultiportPorts m*
and ¬ *has-disc is-Extra m*
shows (*Some sm* = *common-primitive-match-to-simple-match m* ⟶ *matches* (*common-matcher*, α) *m a p* ⟷ *simple-matches sm p*) ∧
(*common-primitive-match-to-simple-match m* = *None* ⟶ ¬ *matches* (*common-matcher*, α) *m a p*)
proof −
show *?thesis*
using *assms* **proof**(*induction m arbitrary*: *sm rule*: *common-primitive-match-to-simple-match.induct*)
case *1* **thus** *?case*

by(*simp add*: *match-iface-simple-match-any-simps bunch-of-lemmata-about-matches simple-matches.simps*)
next
case (*9 p s e*) **thus** *?case*
apply(*simp add*: *match-iface-simple-match-any-simps simple-matches.simps*)
apply(*simp add*: *match-raw-bool ternary-to-bool-bool-to-ternary*)
by *fastforce*
next
case *11* **thus** *?case*
apply(*simp add*: *match-iface-simple-match-any-simps simple-matches.simps*)
apply(*simp add*: *match-raw-bool ternary-to-bool-bool-to-ternary*)
by *fastforce*
next
case (*13 m1 m2*)
let *?caseSome=Some sm* = *common-primitive-match-to-simple-match* (*MatchAnd m1 m2*)
let *?caseNone=common-primitive-match-to-simple-match* (*MatchAnd m1 m2*) = *None*
let *?goal*=(*?caseSome* ⟶ *matches* (*common-matcher*, *α*) (*MatchAnd m1 m2*) *a p* = *simple-matches sm p*) ∧
(*?caseNone* ⟶ ¬ *matches* (*common-matcher*, *α*) (*MatchAnd m1 m2*) *a p*)

from *13* **have** *normalized*:
normalized-src-ports m1 normalized-src-ports m2
normalized-dst-ports m1 normalized-dst-ports m2
normalized-src-ips m1 normalized-src-ips m2
normalized-dst-ips m1 normalized-dst-ips m2
normalized-ifaces m1 normalized-ifaces m2
¬ *has-disc is-L4-Flags m1* ¬ *has-disc is-L4-Flags m2*
¬ *has-disc is-CT-State m1* ¬ *has-disc is-CT-State m2*
¬ *has-disc is-MultiportPorts m1* ¬ *has-disc is-MultiportPorts m2*
¬ *has-disc is-Extra m1* ¬ *has-disc is-Extra m2*
normalized-protocols m1 normalized-protocols m2
by(*simp-all add*: *normalized-protocols-def normalized-ifaces-def*)
{ **assume** *caseNone*: *?caseNone*
{ fix *sm1 sm2*
assume *sm1*: *common-primitive-match-to-simple-match m1* = *Some sm1*
and *sm2*: *common-primitive-match-to-simple-match m2* = *Some sm2*
and *sma*: *simple-match-and sm1 sm2* = *None*
from *sma* **have** *1*: ¬ (*simple-matches sm1 p* ∧ *simple-matches sm2 p*) **by** (*simp add*: *simple-match-and-correct*)
from *normalized sm1 sm2 13.IH* **have** *2*: (*matches* (*common-matcher*, *α*) *m1 a p* ⟷ *simple-matches sm1 p*) ∧
(*matches* (*common-matcher*, *α*) *m2 a p* ⟷ *simple-matches sm2 p*) **by** *force*
hence *2*: *matches* (*common-matcher*, *α*) (*MatchAnd m1 m2*) *a p* ⟷ *simple-matches sm1 p* ∧ *simple-matches sm2 p*
by(*simp add*: *bunch-of-lemmata-about-matches*)

```
      from 1 2 have ¬ matches (common-matcher, α) (MatchAnd m1 m2) a p
by blast
    }
    with caseNone have common-primitive-match-to-simple-match m1 = None
∨
                       common-primitive-match-to-simple-match m2 = None ∨
                       ¬ matches (common-matcher, α) (MatchAnd m1 m2) a p
      by(simp split:option.split-asm)
    hence ¬ matches (common-matcher, α) (MatchAnd m1 m2) a p
      apply(elim disjE)
       apply(simp-all)
       using 13.IH normalized by(simp add: bunch-of-lemmata-about-matches)+
  }note caseNone=this

  { assume caseSome: ?caseSome
     hence ∃ sm1. common-primitive-match-to-simple-match m1 = Some sm1
and
         ∃ sm2. common-primitive-match-to-simple-match m2 = Some sm2
      by(simp-all split: option.split-asm)
   from this obtain sm1 sm2 where sm1: Some sm1 = common-primitive-match-to-simple-match
m1
                      and sm2: Some sm2 = common-primitive-match-to-simple-match
m2 by fastforce+
      with 13.IH normalized have matches (common-matcher, α) m1 a p =
simple-matches sm1 p ∧
                   matches (common-matcher, α) m2 a p = simple-matches sm2 p
by simp
     hence 1: matches (common-matcher, α) (MatchAnd m1 m2) a p ⟷ sim-
ple-matches sm1 p ∧ simple-matches sm2 p
      by(simp add: bunch-of-lemmata-about-matches)
      from caseSome sm1 sm2 have simple-match-and sm1 sm2 = Some sm
by(simp split: option.split-asm)
     hence 2: simple-matches sm p ⟷ simple-matches sm1 p ∧ simple-matches
sm2 p by(simp add: simple-match-and-correct)
     from 1 2 have matches (common-matcher, α) (MatchAnd m1 m2) a p =
simple-matches sm p by simp
  } note caseSome=this

  from caseNone caseSome show ?goal by blast
 qed(simp-all add: match-iface-simple-match-any-simps simple-matches.simps nor-
malized-protocols-def normalized-ifaces-def,
     simp-all add: bunch-of-lemmata-about-matches,
     simp-all add: match-raw-bool ternary-to-bool-bool-to-ternary)
qed

lemma simple-fw-remdups-Rev: simple-fw (remdups-rev rs) p = simple-fw rs p
 apply(induction rs p rule: simple-fw.induct)
  apply(simp add: remdups-rev-def)
 apply(simp-all add: remdups-rev-fst remdups-rev-removeAll simple-fw-not-matches-removeAll)
```

done

fun *action-to-simple-action* :: *action* ⇒ *simple-action* **where**
action-to-simple-action action.Accept = *simple-action.Accept* |
action-to-simple-action action.Drop = *simple-action.Drop* |
action-to-simple-action - = *undefined*

definition *check-simple-fw-preconditions* :: *'i*::*len common-primitive rule list* ⇒ *bool* **where**
check-simple-fw-preconditions rs ≡ ∀ *r* ∈ *set rs*. (*case r of* (*Rule m a*) ⇒
normalized-src-ports m ∧
normalized-dst-ports m ∧
normalized-src-ips m ∧
normalized-dst-ips m ∧
normalized-ifaces m ∧
normalized-protocols m ∧
¬ *has-disc is-L4-Flags m* ∧
¬ *has-disc is-CT-State m* ∧
¬ *has-disc is-MultiportPorts m* ∧
¬ *has-disc is-Extra m* ∧
(*a* = *action.Accept* ∨ *a* = *action.Drop*))

lemma *normalized-src-ports m* ⟹ *normalized-nnf-match m*
apply(*induction m rule*: *normalized-src-ports.induct*)
apply(*simp-all*)[*15*]
oops
lemma ¬ *matcheq-matchNone m* ⟹ *normalized-src-ports m* ⟹ *normalized-nnf-match m*
by(*induction m rule*: *normalized-src-ports.induct*) (*simp-all*)

value *check-simple-fw-preconditions* [*Rule* (*MatchNot* (*MatchNot* (*MatchNot* (*Match* (*Src a*))))) *action.Accept*]

definition *to-simple-firewall* :: *'i*::*len common-primitive rule list* ⇒ *'i simple-rule list* **where**
to-simple-firewall rs ≡ *if check-simple-fw-preconditions rs then*
List.map-filter (λ*r*. *case r of Rule m a* ⇒
(*case* (*common-primitive-match-to-simple-match m*) *of None* ⇒ *None* |
Some sm ⇒ *Some* (*SimpleRule sm* (*action-to-simple-action a*)))) *rs*
else undefined

lemma *to-simple-firewall-simps*:
to-simple-firewall [] = []
check-simple-fw-preconditions ((*Rule m a*)#*rs*) ⟹ *to-simple-firewall* ((*Rule m a*)#*rs*) = (*case common-primitive-match-to-simple-match m of*
None ⇒ *to-simple-firewall rs*

```
  | Some sm ⇒ (SimpleRule sm (action-to-simple-action a)) # to-simple-firewall
rs)
 ¬ check-simple-fw-preconditions rs' ⟹ to-simple-firewall rs' = undefined
 by (auto simp add: to-simple-firewall-def map-filter-def check-simple-fw-preconditions-def
split: option.split)

lemma check-simple-fw-preconditions
    [Rule (MatchAnd (Match (Src (IpAddrNetmask (ipv4addr-of-dotdecimal (127,
0, 0, 0)) 8)))
                    (MatchAnd (Match (Dst-Ports (L4Ports TCP [(0, 65535)])))
                              (Match (Src-Ports (L4Ports TCP [(0, 65535)])))))
           Drop] by eval
lemma to-simple-firewall
    [Rule (MatchAnd (Match (Src (IpAddrNetmask (ipv4addr-of-dotdecimal (127,
0, 0, 0)) 8)))
                    (MatchAnd (Match (Dst-Ports (L4Ports TCP [(0, 65535)])))
                              (Match (Src-Ports (L4Ports TCP [(0, 65535)])))))
           Drop] =
[SimpleRule
  (⦇iiface = Iface ''+'', oiface = Iface ''+'', src = (0x7F000000, 8), dst = (0, 0),
proto = Proto 6, sports = (0, 0xFFFF),
     dports = (0, 0xFFFF)⦈)
  simple-action.Drop] by eval
lemma check-simple-fw-preconditions [Rule (MatchAnd MatchAny MatchAny) Drop]
 by(simp add: check-simple-fw-preconditions-def normalized-ifaces-def normalized-protocols-def)
lemma to-simple-firewall [Rule (MatchAnd MatchAny (MatchAny::32 common-primitive
match-expr)) Drop] =
 [SimpleRule
  (⦇iiface = Iface ''+'', oiface = Iface ''+'', src = (0, 0), dst = (0, 0), proto =
ProtoAny, sports = (0, 0xFFFF),
     dports = (0, 0xFFFF)⦈)
  simple-action.Drop] by eval
lemma to-simple-firewall [Rule (Match (Src (IpAddrNetmask (ipv4addr-of-dotdecimal
(127, 0, 0, 0)) 8))) Drop] =
[SimpleRule
  (⦇iiface = Iface ''+'', oiface = Iface ''+'', src = (0x7F000000, 8), dst = (0, 0),
proto = ProtoAny, sports = (0, 0xFFFF),
     dports = (0, 0xFFFF)⦈)
  simple-action.Drop] by eval



theorem to-simple-firewall: check-simple-fw-preconditions rs ⟹ approximating-bigstep-fun
(common-matcher, α) p rs Undecided = simple-fw (to-simple-firewall rs) p
 proof(induction rs)
 case Nil thus ?case by(simp add: to-simple-firewall-simps)
 next
 case (Cons r rs)
   from Cons have IH: approximating-bigstep-fun (common-matcher, α) p rs
```

Undecided = *simple-fw* (*to-simple-firewall rs*) *p*
by(*simp add*: *check-simple-fw-preconditions-def*)
obtain *m a* **where** *r*: *r* = *Rule m a* **by**(*cases r*, *simp*)
from *Cons.prems* **have** *check-simple-fw-preconditions* [*r*] **by**(*simp add*: *check-simple-fw-preconditions-def*)
with *r common-primitive-match-to-simple-match*[**where** *p* = *p*]
have *match*: ⋀ *sm*. *common-primitive-match-to-simple-match m* = *Some sm* ⟹ *matches* (*common-matcher*, α) *m a p* = *simple-matches sm p* **and**
nomatch: *common-primitive-match-to-simple-match m* = *None* ⟹ ¬ *matches* (*common-matcher*, α) *m a p*
unfolding *check-simple-fw-preconditions-def* **by** *simp-all*
from *to-simple-firewall-simps r Cons.prems* **have** *to-simple-firewall-simps'*: *to-simple-firewall* (*Rule m a* # *rs*) =
(*case common-primitive-match-to-simple-match m of None* ⇒ *to-simple-firewall rs*
| *Some sm* ⇒ *SimpleRule sm* (*action-to-simple-action a*) # *to-simple-firewall rs*) **by** *simp*
from ‹*check-simple-fw-preconditions* [*r*]› **have** *a* = *action.Accept* ∨ *a* = *action.Drop* **by**(*simp add*: *r check-simple-fw-preconditions-def*)
thus *?case*
by(*auto simp add*: *r to-simple-firewall-simps' IH match nomatch split*: *option.split action.split*)
qed

lemma *ctstate-assume-new-not-has-CT-State*:
r ∈ *set* (*ctstate-assume-new rs*) ⟹ ¬ *has-disc is-CT-State* (*get-match r*)
apply(*simp add*: *ctstate-assume-new-def*)
apply(*induction rs*)
apply(*simp add*: *optimize-matches-def*; *fail*)
apply(*simp add*: *optimize-matches-def*)
apply(*rename-tac r' rs*, *case-tac r'*)
apply(*safe*)
apply(*simp add*: *split*:*if-split-asm*)
apply(*elim disjE*)
apply(*simp-all add*: *not-hasdisc-ctstate-assume-state split*:*if-split-asm*)
done

The precondition for the simple firewall can be easily fulfilled. The subset relation is due to abstracting over some primitives (e.g., negated primitives, l4 flags)

theorem *transform-simple-fw-upper*:
defines *preprocess rs* ≡ *upper-closure* (*optimize-matches abstract-for-simple-firewall* (*upper-closure* (*packet-assume-new rs*)))
and *newpkt p* ≡ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*
assumes *simplers*: *simple-ruleset* (*rs*:: 'i::*len common-primitive rule list*)
— the preconditions for the simple firewall are fulfilled, definitely no runtime failure
shows *check-simple-fw-preconditions* (*preprocess rs*)

— the set of new packets, which are accepted is an overapproximations
and {*p*. (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} ⊆
{*p*. *simple-fw* (*to-simple-firewall* (*preprocess rs*)) *p* = *Decision FinalAllow* ∧ *newpkt p*}
— Fun fact: The theorem holds for a tagged packet. The simple firewall just ignores the tag. You may explicitly untag, if you wish to, but a *'i tagged-packet* is just an extension of the *'i simple-packet* used by the simple firewall
unfolding *check-simple-fw-preconditions-def preprocess-def*
apply(*clarify*, *rename-tac r*, *case-tac r*, *rename-tac m a*, *simp*)
proof −
let *?rs2*=*upper-closure* (*packet-assume-new rs*)
let *?rs3*=*optimize-matches abstract-for-simple-firewall ?rs2*
let *?rs'*=*upper-closure ?rs3*
let *?γ*=(*common-matcher*, *in-doubt-allow*)
:: ('*i*::*len common-primitive*, ('*i*, '*a*) *tagged-packet-scheme*) *match-tac*
let *?fw*=λ*rs p*. *approximating-bigstep-fun ?γ p rs Undecided*

from *packet-assume-new-simple-ruleset*[*OF simplers*] **have** *s1*: *simple-ruleset* (*packet-assume-new rs*) **.**
from *transform-upper-closure*(*2*)[*OF s1*] **have** *s2*: *simple-ruleset ?rs2* **.**
from *s2* **have** *s3*: *simple-ruleset ?rs3* **by** (*simp add*: *optimize-matches-simple-ruleset*)

from *transform-upper-closure*(*2*)[*OF s3*] **have** *s4*: *simple-ruleset ?rs'* **.**

from *transform-upper-closure*(*3*)[*OF s1*] **have** *nnf2*:
∀ *r*∈*set* (*upper-closure* (*packet-assume-new rs*)). *normalized-nnf-match* (*get-match r*) **by** *simp*

{ fix *m a*
assume *r*: *Rule m a* ∈ *set ?rs'*

from *s4 r* **have** *a*: (*a* = *action.Accept* ∨ *a* = *action.Drop*) **by**(*auto simp add*: *simple-ruleset-def*)

have *r* ∈ *set* (*packet-assume-new rs*) ⟹ ¬ *has-disc is-CT-State* (*get-match r*) **for** *r*
by(*simp add*: *packet-assume-new-def ctstate-assume-new-not-has-CT-State*)
with *transform-upper-closure*(*4*)[*OF s1*, **where** *disc*=*is-CT-State*] **have**
∀ *r*∈*set* (*upper-closure* (*packet-assume-new rs*)). ¬ *has-disc is-CT-State* (*get-match r*)
by *simp*
with *abstract-primitive-preserves-nodisc*[**where** *disc'*=*is-CT-State*]
have ∀ *r*∈*set ?rs3*. ¬ *has-disc is-CT-State* (*get-match r*)
apply(*intro optimize-matches-preserves*)
by(*auto simp add*: *abstract-for-simple-firewall-def*)
with *transform-upper-closure*(*4*)[*OF s3*, **where** *disc*=*is-CT-State*] **have**
∀ *r*∈*set ?rs'*. ¬ *has-disc is-CT-State* (*get-match r*) **by** *simp*
with *r* **have** *no-CT*: ¬ *has-disc is-CT-State m* **by** *fastforce*

from *abstract-for-simple-firewall-hasdisc* **have** $\forall$ *r*$\in$*set ?rs3*. $\neg$ *has-disc is-L4-Flags* (*get-match r*)
by(*intro optimize-matches-preserves*, *auto*)
with *transform-upper-closure*(*4*)[*OF s3*, **where** *disc*=*is-L4-Flags*] **have**
$\forall$ *r*$\in$*set ?rs'*. $\neg$ *has-disc is-L4-Flags* (*get-match r*) **by** *simp*
with *r* **have** *no-L4-Flags*: $\neg$ *has-disc is-L4-Flags m* **by** *fastforce*

from *nnf2 abstract-for-simple-firewall-negated-ifaces-prots* **have**
ifaces: $\forall$ *r*$\in$*set ?rs3*. $\neg$ *has-disc-negated* (λ*a. is-Iiface a* $\vee$ *is-Oiface a*) *False* (*get-match r*) **and**
protocols-rs3: $\forall$ *r*$\in$*set ?rs3*. $\neg$ *has-disc-negated is-Prot False* (*get-match r*)
by(*intro optimize-matches-preserves*, *blast*)+
from *ifaces* **have** *iface-in*: $\forall$ *r*$\in$*set ?rs3*. $\neg$ *has-disc-negated is-Iiface False* (*get-match r*) **and**
iface-out: $\forall$ *r*$\in$*set ?rs3*. $\neg$ *has-disc-negated is-Oiface False* (*get-match r*)
using *has-disc-negated-disj-split* **by** *blast*+

from *transform-upper-closure*(*3*)[*OF s3*] **have** $\forall$ *r*$\in$*set ?rs'*.
normalized-nnf-match (*get-match r*) $\wedge$ *normalized-src-ports* (*get-match r*) $\wedge$
normalized-dst-ports (*get-match r*) $\wedge$ *normalized-src-ips* (*get-match r*) $\wedge$
normalized-dst-ips (*get-match r*) $\wedge$
$\neg$ *has-disc is-MultiportPorts* (*get-match r*) $\wedge$ $\neg$ *has-disc is-Extra* (*get-match r*)
.
with *r* **have** *normalized*:
normalized-src-ports m $\wedge$ *normalized-dst-ports m* $\wedge$
normalized-src-ips m $\wedge$ *normalized-dst-ips m* $\wedge$
$\neg$ *has-disc is-MultiportPorts m* & $\neg$ *has-disc is-Extra m* **by** *fastforce*

from *transform-upper-closure*(*5*)[*OF s3*] *iface-in iface-out* **have** $\forall$ *r*$\in$*set ?rs'*.
$\neg$ *has-disc-negated is-Iiface False* (*get-match r*) $\wedge$ $\neg$ *has-disc-negated is-Oiface False* (*get-match r*) **by** *simp*
with *r* **have** *abstracted-ifaces*: *normalized-ifaces m*
unfolding *normalized-ifaces-def has-disc-negated-disj-split* **by** *fastforce*

from *transform-upper-closure*(*3*)[*OF s1*]
normalized-n-primitive-imp-not-disc-negated[*OF wf-disc-sel-common-primitive*(*1*)]
normalized-n-primitive-imp-not-disc-negated[*OF wf-disc-sel-common-primitive*(*2*)]
have $\forall$ *r*$\in$ *set ?rs2*. $\neg$ *has-disc-negated is-Src-Ports False* (*get-match r*) $\wedge$
$\neg$ *has-disc-negated is-Dst-Ports False* (*get-match r*) $\wedge$
$\neg$ *has-disc is-MultiportPorts* (*get-match r*)
apply(*simp add*: *normalized-src-ports-def2 normalized-dst-ports-def2*)
by *blast*
from *this* **have** $\forall$ *r*$\in$*set ?rs3*. $\neg$ *has-disc-negated is-Src-Ports False* (*get-match r*) $\wedge$
$\neg$ *has-disc-negated is-Dst-Ports False* (*get-match r*) $\wedge$
$\neg$ *has-disc is-MultiportPorts* (*get-match r*)

apply −
apply(*rule optimize-matches-preserves*)
apply(*intro conjI*)
apply(*intro abstract-for-simple-firewall-preserves-nodisc-negated*, *simp-all*)+
by (*simp add*: *abstract-for-simple-firewall-def abstract-primitive-preserves-nodisc*)

from *this protocols-rs3 transform-upper-closure*(*5*)[*OF s3*, **where** *disc*=*is-Prot*, *simplified*]
have ∀ *r*∈*set ?rs′*. ¬ *has-disc-negated is-Prot False* (*get-match r*)
by *simp*
with *r* **have** *abstracted-prots*: *normalized-protocols m*
unfolding *normalized-protocols-def has-disc-negated-disj-split* **by** *fastforce*

from *no-CT no-L4-Flags s4 normalized a abstracted-ifaces abstracted-prots*
show *normalized-src-ports m* ∧
normalized-dst-ports m ∧
normalized-src-ips m ∧
normalized-dst-ips m ∧
normalized-ifaces m ∧
normalized-protocols m ∧
¬ *has-disc is-L4-Flags m* ∧
¬ *has-disc is-CT-State m* ∧
¬ *has-disc is-MultiportPorts m* ∧
¬ *has-disc is-Extra m* ∧ (*a* = *action.Accept* ∨ *a* = *action.Drop*)
by(*simp*)
}
hence *simple-fw-preconditions*: *check-simple-fw-preconditions ?rs′*
unfolding *check-simple-fw-preconditions-def*
by(*clarify*, *rename-tac r*, *case-tac r*, *rename-tac m a*, *simp*)

have *1*: {*p*. *?γ*,*p*⊢ ⟨*?rs′*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} =
{*p*. *?γ*,*p*⊢ ⟨*?rs3*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
apply(*subst transform-upper-closure*(*1*)[*OF s3*])
by *simp*
from *abstract-primitive-in-doubt-allow-generic*(*2*)[*OF primitive-matcher-generic-common-matcher nnf2 s2*] **have** *2*:
{*p*. *?γ*,*p*⊢ ⟨*upper-closure* (*packet-assume-new rs*), *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} ⊆
{*p*. *?γ*,*p*⊢ ⟨*?rs3*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
by(*auto simp add*: *abstract-for-simple-firewall-def*)
have *3*: {*p*. *?γ*,*p*⊢ ⟨*upper-closure* (*packet-assume-new rs*), *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} =
{*p*. *?γ*,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
apply(*subst transform-upper-closure*(*1*)[*OF s1*])
apply(*subst approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF s1*]])
apply(*subst approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers*]])

using *packet-assume-new newpkt-def* **by** *fastforce*

have *4*: $\bigwedge p$. *?γ,p⊢* ⟨*?rs′*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ⟷ *?fw ?rs′ p* = *Decision FinalAllow*
using *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF s4*]] **by** *fast*

have {*p*. *?γ,p⊢* ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} ⊆
{*p*. *?γ,p⊢* ⟨*?rs′*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
apply(*subst 1*)
apply(*subst 3*[*symmetric*])
using *2* **by** *blast*

thus {*p*. *?γ,p⊢* ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} ⊆
{*p*. *simple-fw* (*to-simple-firewall ?rs′*) *p* = *Decision FinalAllow* ∧ *newpkt p*}
apply *safe*
subgoal for *p* **using** *to-simple-firewall*[*OF simple-fw-preconditions*, **where** *p* = *p*] *4* **by** *auto*
done
qed

theorem *transform-simple-fw-lower*:
defines *preprocess rs* ≡ *lower-closure* (*optimize-matches abstract-for-simple-firewall* (*lower-closure* (*packet-assume-new rs*)))
and *newpkt p* ≡ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*
assumes *simplers*: *simple-ruleset* (*rs*:: *′i*::*len common-primitive rule list*)
— the preconditions for the simple firewall are fulfilled, definitely no runtime failure
shows *check-simple-fw-preconditions* (*preprocess rs*)
— the set of new packets, which are accepted is an underapproximation
and {*p*. *simple-fw* (*to-simple-firewall* (*preprocess rs*)) *p* = *Decision FinalAllow* ∧ *newpkt p*} ⊆
{*p*. (*common-matcher*, *in-doubt-deny*),*p⊢* ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
unfolding *check-simple-fw-preconditions-def preprocess-def*
apply(*clarify*, *rename-tac r*, *case-tac r*, *rename-tac m a*, *simp*)
proof −
let *?rs2*=*lower-closure* (*packet-assume-new rs*)
let *?rs3*=*optimize-matches abstract-for-simple-firewall ?rs2*
let *?rs′*=*lower-closure ?rs3*
let *?γ*=(*common-matcher*, *in-doubt-deny*)
:: (*′i*::*len common-primitive*, (*′i*, *′a*) *tagged-packet-scheme*) *match-tac*
let *?fw*=λ*rs p*. *approximating-bigstep-fun ?γ p rs Undecided*

from *packet-assume-new-simple-ruleset*[*OF simplers*] **have** *s1*: *simple-ruleset* (*packet-assume-new rs*) **.**

from *transform-lower-closure*(*2*)[*OF s1*] **have** *s2*: *simple-ruleset* (*lower-closure* (*packet-assume-new rs*)) **.**
from *s2* **have** *s3*: *simple-ruleset ?rs3* **by** (*simp add*: *optimize-matches-simple-ruleset*)

from *transform-lower-closure*(*2*)[*OF s3*] **have** *s4*: *simple-ruleset ?rs′* **.**

from *transform-lower-closure*(*3*)[*OF s1*] **have** *nnf2*:
∀ *r*∈*set* (*lower-closure* (*packet-assume-new rs*)). *normalized-nnf-match* (*get-match r*) **by** *simp*

{ fix *m a*
assume *r*: *Rule m a* ∈ *set ?rs′*

from *s4 r* **have** *a*: (*a* = *action.Accept* ∨ *a* = *action.Drop*) **by**(*auto simp add*: *simple-ruleset-def*)

have *r* ∈ *set* (*packet-assume-new rs*) ⟹ ¬ *has-disc is-CT-State* (*get-match r*) **for** *r*
by(*simp add*: *packet-assume-new-def ctstate-assume-new-not-has-CT-State*)
with *transform-lower-closure*(*4*)[*OF s1*, **where** *disc*=*is-CT-State*] **have**
∀ *r*∈*set* (*lower-closure* (*packet-assume-new rs*)). ¬ *has-disc is-CT-State* (*get-match r*)
by *fastforce*
with *abstract-primitive-preserves-nodisc*[**where** *disc′*=*is-CT-State*] **have**
∀ *r*∈*set ?rs3*. ¬ *has-disc is-CT-State* (*get-match r*)
apply(*intro optimize-matches-preserves*)
by(*auto simp add*: *abstract-for-simple-firewall-def*)
with *transform-lower-closure*(*4*)[*OF s3*, **where** *disc*=*is-CT-State*] **have**
∀ *r*∈*set ?rs′*. ¬ *has-disc is-CT-State* (*get-match r*) **by** *fastforce*
with *r* **have** *no-CT*: ¬ *has-disc is-CT-State m* **by** *fastforce*

from *abstract-for-simple-firewall-hasdisc* **have** ∀ *r*∈*set ?rs3*. ¬ *has-disc is-L4-Flags* (*get-match r*)
by(*intro optimize-matches-preserves*, *blast*)
with *transform-lower-closure*(*4*)[*OF s3*, **where** *disc*=*is-L4-Flags*] **have**
∀ *r*∈*set ?rs′*. ¬ *has-disc is-L4-Flags* (*get-match r*) **by** *fastforce*
with *r* **have** *no-L4-Flags*: ¬ *has-disc is-L4-Flags m* **by** *fastforce*

from *nnf2 abstract-for-simple-firewall-negated-ifaces-prots* **have**
ifaces: ∀ *r*∈*set ?rs3*. ¬ *has-disc-negated* (λ*a*. *is-Iiface a* ∨ *is-Oiface a*) *False* (*get-match r*) **and**
protocols-rs3: ∀ *r*∈*set ?rs3*. ¬ *has-disc-negated is-Prot False* (*get-match r*)
by(*intro optimize-matches-preserves*, *blast*)+
from *ifaces* **have** *iface-in*: ∀ *r*∈*set ?rs3*. ¬ *has-disc-negated is-Iiface False* (*get-match r*) **and**
iface-out: ∀ *r*∈*set ?rs3*. ¬ *has-disc-negated is-Oiface False* (*get-match r*)
using *has-disc-negated-disj-split* **by** *blast*+

from *transform-lower-closure*(*3*)[*OF s3*] **have** $\forall r{\in}$*set ?rs'*.
normalized-nnf-match (*get-match r*) $\wedge$ *normalized-src-ports* (*get-match r*) $\wedge$
normalized-dst-ports (*get-match r*) $\wedge$ *normalized-src-ips* (*get-match r*) $\wedge$
normalized-dst-ips (*get-match r*) $\wedge$
$\neg$ *has-disc is-MultiportPorts* (*get-match r*) $\wedge$ $\neg$ *has-disc is-Extra* (*get-match r*)
.
with *r* **have** *normalized*: *normalized-src-ports m* $\wedge$ *normalized-dst-ports m* $\wedge$ *normalized-src-ips m* $\wedge$
normalized-dst-ips m $\wedge$ $\neg$ *has-disc is-MultiportPorts m* $\wedge$ $\neg$ *has-disc is-Extra m* **by** *fastforce*

from *transform-lower-closure*(*5*)[*OF s3*] *iface-in iface-out* **have** $\forall r{\in}$*set ?rs'*.
$\neg$ *has-disc-negated is-Iiface False* (*get-match r*) $\wedge$ $\neg$ *has-disc-negated is-Oiface False* (*get-match r*) **by** *simp*
with *r* **have** *abstracted-ifaces*: *normalized-ifaces m*
unfolding *normalized-ifaces-def has-disc-negated-disj-split* **by** *fastforce*

from *transform-lower-closure*(*3*)[*OF s1*]
normalized-n-primitive-imp-not-disc-negated[*OF wf-disc-sel-common-primitive*(*1*)]
normalized-n-primitive-imp-not-disc-negated[*OF wf-disc-sel-common-primitive*(*2*)]
have $\forall r{\in}$*set ?rs2*. $\neg$ *has-disc-negated is-Src-Ports False* (*get-match r*) $\wedge$
$\neg$ *has-disc-negated is-Dst-Ports False* (*get-match r*) $\wedge$
$\neg$ *has-disc is-MultiportPorts* (*get-match r*)
apply(*simp add*: *normalized-src-ports-def2 normalized-dst-ports-def2*)
by *blast*
from *this* **have** $\forall r{\in}$*set ?rs3*. $\neg$ *has-disc-negated is-Src-Ports False* (*get-match r*) $\wedge$
$\neg$ *has-disc-negated is-Dst-Ports False* (*get-match r*) $\wedge$
$\neg$ *has-disc is-MultiportPorts* (*get-match r*)
apply $-$
apply(*rule optimize-matches-preserves*)
apply(*intro conjI*)
apply(*intro abstract-for-simple-firewall-preserves-nodisc-negated*, *simp-all*)+
by (*simp add*: *abstract-for-simple-firewall-def abstract-primitive-preserves-nodisc*)
from *this protocols-rs3 transform-lower-closure*(*5*)[*OF s3*, **where** *disc*=*is-Prot*, *simplified*]
have $\forall r{\in}$*set ?rs'*. $\neg$ *has-disc-negated is-Prot False* (*get-match r*)
by *simp*
with *r* **have** *abstracted-prots*: *normalized-protocols m*
unfolding *normalized-protocols-def has-disc-negated-disj-split* **by** *fastforce*

from *no-CT no-L4-Flags s4 normalized a abstracted-ifaces abstracted-prots*
show *normalized-src-ports m* $\wedge$
normalized-dst-ports m $\wedge$
normalized-src-ips m $\wedge$
normalized-dst-ips m $\wedge$
normalized-ifaces m $\wedge$
normalized-protocols m $\wedge$ $\neg$ *has-disc is-L4-Flags m* $\wedge$ $\neg$ *has-disc*

is-CT-State m ∧
¬ *has-disc is-MultiportPorts m* ∧ ¬ *has-disc is-Extra m* ∧ (*a* = *action.Accept* ∨ *a* = *action.Drop*)
by(*simp*)
}
hence *simple-fw-preconditions*: *check-simple-fw-preconditions ?rs′*
unfolding *check-simple-fw-preconditions-def*
by(*clarify*, *rename-tac r*, *case-tac r*, *rename-tac m a*, *simp*)

have *1*: {*p*. *?γ*,*p*⊢ ⟨*?rs′*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} =
{*p*. *?γ*,*p*⊢ ⟨*?rs3*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
apply(*subst transform-lower-closure*(*1*)[*OF s3*])
by *simp*
from *abstract-primitive-in-doubt-deny-generic*(*1*)[*OF primitive-matcher-generic-common-matcher nnf2 s2*] **have** *2*:
{*p*. *?γ*,*p*⊢ ⟨*?rs3*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} ⊆
{*p*. *?γ*,*p*⊢ ⟨*lower-closure* (*packet-assume-new rs*), *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
by(*auto simp add*: *abstract-for-simple-firewall-def*)
have *3*: {*p*. *?γ*,*p*⊢ ⟨*lower-closure* (*packet-assume-new rs*), *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} =
{*p*. *?γ*,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
apply(*subst transform-lower-closure*(*1*)[*OF s1*])
apply(*subst approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF s1*]])
apply(*subst approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers*]])
using *packet-assume-new newpkt-def* **by** *fastforce*

have *4*: ⋀*p*. *?γ*,*p*⊢ ⟨*?rs′*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ⟷ *?fw ?rs′ p* = *Decision FinalAllow*
using *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF s4*]] **by** *fast*

have {*p*. *?γ*,*p*⊢ ⟨*?rs′*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} ⊆
{*p*. *?γ*,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
apply(*subst 1*)
apply(*subst 3*[*symmetric*])
using *2* **by** *blast*

thus {*p*. *simple-fw* (*to-simple-firewall ?rs′*) *p* = *Decision FinalAllow* ∧ *newpkt p*} ⊆
{*p*. *?γ*,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
apply *safe*
subgoal for *p* **using** *to-simple-firewall*[*OF simple-fw-preconditions*, **where** *p* = *p*] *4* **by** *auto*
done
qed

definition *to-simple-firewall-without-interfaces ipassmt rtblo rs* ≡
to-simple-firewall
(*upper-closure*
(*optimize-matches* (*abstract-primitive* (λ*r*. *case r of Pos a* ⇒ *is-Iiface a* ∨ *is-Oiface a* | *Neg a* ⇒ *is-Iiface a* ∨ *is-Oiface a*))
(*optimize-matches abstract-for-simple-firewall*
(*upper-closure*
(*iface-try-rewrite ipassmt rtblo*
(*upper-closure*
(*packet-assume-new rs*)))))))

theorem *to-simple-firewall-without-interfaces*:
defines *newpkt p* ≡ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*
assumes *simplers*: *simple-ruleset* (*rs*:: *'i*::*len common-primitive rule list*)

— well-formed ipassmt
and *wf-ipassmt1*: *ipassmt-sanity-nowildcards* (*map-of ipassmt*) **and** *wf-ipassmt2*: *distinct* (*map fst ipassmt*)
— There are no spoofed packets (probably by kernel's reverse path filter or our checker). This assumption implies that ipassmt lists ALL interfaces (!!).
and *nospoofing*: ∀ (*p*::(*'i*::*len*, *'a*) *tagged-packet-scheme*).
∃ *ips*. (*map-of ipassmt*) (*Iface* (*p-iiface p*)) = *Some ips* ∧ *p-src p* ∈ *ipcidr-union-set* (*set ips*)
— If a routing table was passed, the output interface for any packet we consider is decided based on it.
and *routing-decided*: ⋀*rtbl* (*p*::(*'i*,*'a*) *tagged-packet-scheme*). *rtblo* = *Some rtbl* ⟹ *output-iface* (*routing-table-semantics rtbl* (*p-dst p*)) = *p-oiface p*
— A passed routing table is wellformed
and *correct-routing*: ⋀*rtbl*. *rtblo* = *Some rtbl* ⟹ *correct-routing rtbl*
— A passed routing table contains no interfaces with wildcard names
and *routing-no-wildcards*: ⋀*rtbl*. *rtblo* = *Some rtbl* ⟹ *ipassmt-sanity-nowildcards* (*map-of* (*routing-ipassmt rtbl*))

— the set of new packets, which are accepted is an overapproximations
shows {*p*::(*'i*,*'a*) *tagged-packet-scheme*. (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} ⊆
{*p*::(*'i*,*'a*) *tagged-packet-scheme*. *simple-fw* (*to-simple-firewall-without-interfaces ipassmt rtblo rs*) *p* = *Decision FinalAllow* ∧ *newpkt p*}

and ∀ *r* ∈ *set* (*to-simple-firewall-without-interfaces ipassmt rtblo rs*).
iiface (*match-sel r*) = *ifaceAny* ∧ *oiface* (*match-sel r*) = *ifaceAny*
proof −
let *?rs1*=*packet-assume-new rs*
let *?rs2*=*upper-closure ?rs1*

let *?rs3=iface-try-rewrite ipassmt rtblo ?rs2*
let *?rs4=upper-closure ?rs3*
let *?rs5=optimize-matches abstract-for-simple-firewall ?rs4*
let *?rs6=optimize-matches (abstract-primitive (λr. case r of Pos a ⇒ is-Iiface a ∨ is-Oiface a | Neg a ⇒ is-Iiface a ∨ is-Oiface a)) ?rs5*
let *?rs7=upper-closure ?rs6*
let *?γ=(common-matcher, in-doubt-allow)*
:: *('i::len common-primitive, ('i, 'a) tagged-packet-scheme) match-tac*

have *to-simple-firewall-without-interfaces ipassmt rtblo rs = to-simple-firewall ?rs7*
by(*simp add*: *to-simple-firewall-without-interfaces-def*)

from *packet-assume-new-simple-ruleset*[*OF simplers*] **have** *s1*: *simple-ruleset ?rs1* **.**
from *transform-upper-closure*(*2*)[*OF s1*] **have** *s2*: *simple-ruleset ?rs2* **.**
from *iface-try-rewrite-simplers*[*OF s2*] **have** *s3*: *simple-ruleset ?rs3* **.**
from *transform-upper-closure*(*2*)[*OF s3*] **have** *s4*: *simple-ruleset ?rs4* **.**
from *optimize-matches-simple-ruleset*[*OF s4*] **have** *s5*: *simple-ruleset ?rs5* **.**
from *optimize-matches-simple-ruleset*[*OF s5*] **have** *s6*: *simple-ruleset ?rs6* **.**
from *transform-upper-closure*(*2*)[*OF s6*] **have** *s7*: *simple-ruleset ?rs7* **.**

from *transform-upper-closure*(*3*)[*OF s1*] **have** *nnf2*: ∀ *r*∈*set ?rs2*. *normalized-nnf-match* (*get-match r*) **by** *simp*
from *transform-upper-closure*(*3*)[*OF s3*] **have** *nnf4*: ∀ *r*∈*set ?rs4*. *normalized-nnf-match* (*get-match r*) **by** *simp*
have *nnf5*: ∀ *r*∈*set ?rs5*. *normalized-nnf-match* (*get-match r*)
apply(*intro optimize-matches-preserves*)
apply(*simp add*: *abstract-for-simple-firewall-def*)
apply(*rule abstract-primitive-preserves-normalized*(*5*))
using *nnf4* **by**(*simp*)
have *nnf6*: ∀ *r*∈*set ?rs6*. *normalized-nnf-match* (*get-match r*)
apply(*intro optimize-matches-preserves*)
apply(*rule abstract-primitive-preserves-normalized*(*5*))
using *nnf5* **by**(*simp*)
from *transform-upper-closure*(*3*)[*OF s6*] **have** *nnf7*: ∀ *r*∈*set ?rs7*. *normalized-nnf-match* (*get-match r*) **by** *simp*

{ fix *m a*
assume *r*: *Rule m a* ∈ *set ?rs7*

from *s7 r* **have** *a*: (*a* = *action.Accept* ∨ *a* = *action.Drop*) **by**(*auto simp add*: *simple-ruleset-def*)

from *abstract-for-simple-firewall-hasdisc* **have** ∀ *r*∈*set ?rs5*. ¬ *has-disc is-CT-State* (*get-match r*)
by(*intro optimize-matches-preserves*, *blast*)

with *abstract-primitive-preserves-nodisc*[**where** *disc′=is-CT-State*] **have**
∀ *r*∈*set ?rs6*. ¬ *has-disc is-CT-State* (*get-match r*)
apply(*intro optimize-matches-preserves*)
apply(*simp*)
by *blast*
with *transform-upper-closure*(*4*)[*OF s6*, **where** *disc=is-CT-State*] **have**
∀ *r*∈*set ?rs7*. ¬ *has-disc is-CT-State* (*get-match r*) **by** *simp*
with *r* **have** *no-CT*: ¬ *has-disc is-CT-State m* **by** *fastforce*

from *abstract-for-simple-firewall-hasdisc* **have** ∀ *r*∈*set ?rs5*. ¬ *has-disc is-L4-Flags* (*get-match r*)
by(*intro optimize-matches-preserves*, *blast*)
with *abstract-primitive-preserves-nodisc*[**where** *disc′=is-L4-Flags*] **have**
∀ *r*∈*set ?rs6*. ¬ *has-disc is-L4-Flags* (*get-match r*)
by(*intro optimize-matches-preserves*) *auto*
with *transform-upper-closure*(*4*)[*OF s6*, **where** *disc=is-L4-Flags*] **have**
∀ *r*∈*set ?rs7*. ¬ *has-disc is-L4-Flags* (*get-match r*) **by** *simp*
with *r* **have** *no-L4-Flags*: ¬ *has-disc is-L4-Flags m* **by** *fastforce*

have ∀ *r*∈*set ?rs6*. ¬ *has-disc is-Iiface* (*get-match r*)
by(*intro optimize-matches-preserves abstract-primitive-nodisc*) *simp*+
with *transform-upper-closure*(*4*)[*OF s6*, **where** *disc=is-Iiface*] **have**
∀ *r*∈*set ?rs7*. ¬ *has-disc is-Iiface* (*get-match r*) **by** *simp*
with *r* **have** *no-Iiface*: ¬ *has-disc is-Iiface m* **by** *fastforce*

have ∀ *r*∈*set ?rs6*. ¬ *has-disc is-Oiface* (*get-match r*)
by(*intro optimize-matches-preserves abstract-primitive-nodisc*) *simp*+
with *transform-upper-closure*(*4*)[*OF s6*, **where** *disc=is-Oiface*] **have**
∀ *r*∈*set ?rs7*. ¬ *has-disc is-Oiface* (*get-match r*) **by** *simp*
with *r* **have** *no-Oiface*: ¬ *has-disc is-Oiface m* **by** *fastforce*

from *no-Iiface no-Oiface* **have** *normalized-ifaces*: *normalized-ifaces m*
using *has-disc-negated-disj-split has-disc-negated-has-disc normalized-ifaces-def*
by *blast*

from *transform-upper-closure*(*3*)[*OF s6*] *r* **have** *normalized*:
normalized-src-ports m ∧ *normalized-dst-ports m* ∧
normalized-src-ips m ∧ *normalized-dst-ips m* ∧
¬ *has-disc is-MultiportPorts m* ∧ ¬ *has-disc is-Extra m* **by** *fastforce*

from *transform-upper-closure*(*3*)[*OF s3*, *simplified*]
normalized-n-primitive-imp-not-disc-negated[*OF wf-disc-sel-common-primitive*(*1*)]
normalized-n-primitive-imp-not-disc-negated[*OF wf-disc-sel-common-primitive*(*2*)]
have ∀ *r* ∈ *set ?rs4*. ¬ *has-disc-negated is-Src-Ports False* (*get-match r*) ∧
¬ *has-disc-negated is-Dst-Ports False* (*get-match r*) ∧
¬ *has-disc is-MultiportPorts* (*get-match r*)
apply(*simp add*: *normalized-src-ports-def2 normalized-dst-ports-def2*)

```
      by blast
    hence ∀ r ∈ set ?rs5. ¬ has-disc-negated is-Src-Ports False (get-match r) ∧
                          ¬ has-disc-negated is-Dst-Ports False (get-match r) ∧
                          ¬ has-disc is-MultiportPorts (get-match r)
      apply −
      apply(rule optimize-matches-preserves)
      apply(intro conjI)
     apply(intro abstract-for-simple-firewall-preserves-nodisc-negated, simp-all)+
    by (simp add: abstract-for-simple-firewall-def abstract-primitive-preserves-nodisc)
    from this have no-ports-rs6:
        ∀ r ∈ set ?rs6. ¬ has-disc-negated is-Src-Ports False (get-match r) ∧
                        ¬ has-disc-negated is-Dst-Ports False (get-match r) ∧
                        ¬ has-disc is-MultiportPorts (get-match r)
      apply −
      apply(rule optimize-matches-preserves)
      apply(intro conjI)
       apply(intro abstract-primitive-preserves-nodisc-nedgated, simp-all)+
    by (simp add: abstract-for-simple-firewall-def abstract-primitive-preserves-nodisc)

    from nnf4 abstract-for-simple-firewall-negated-ifaces-prots(2) have
     ∀ r∈set ?rs5. ¬ has-disc-negated is-Prot False (get-match r)
     by(intro optimize-matches-preserves) blast
    hence ∀ r∈set ?rs6. ¬ has-disc-negated is-Prot False (get-match r)
    by(intro optimize-matches-preserves abstract-primitive-preserves-nodisc-nedgated)
simp+
   with no-ports-rs6 have ∀ r∈set ?rs7. ¬ has-disc-negated is-Prot False (get-match
r)
     by(intro transform-upper-closure(5)[OF s6]) simp+
   with r have protocols: normalized-protocols m unfolding normalized-protocols-def
by fastforce


     from no-CT no-L4-Flags normalized a normalized-ifaces protocols no-Iiface
no-Oiface
      have normalized-src-ports m ∧
           normalized-dst-ports m ∧
           normalized-src-ips m ∧
           normalized-dst-ips m ∧
           normalized-ifaces m ∧
           normalized-protocols m ∧
           ¬ has-disc is-L4-Flags m ∧
           ¬ has-disc is-CT-State m ∧
           ¬ has-disc is-MultiportPorts m ∧
           ¬ has-disc is-Extra m ∧ (a = action.Accept ∨ a = action.Drop)
     and ¬ has-disc is-Iiface m and ¬ has-disc is-Oiface m
     apply −
     by(simp)+
   }
   hence simple-fw-preconditions: check-simple-fw-preconditions ?rs7
```

and *no-interfaces*: *Rule m a* ∈ *set ?rs7* ⟹ ¬ *has-disc is-Iiface m* ∧ ¬ *has-disc is-Oiface m* **for** *m a*
apply −
subgoal unfolding *check-simple-fw-preconditions-def* **by**(*clarify*, *rename-tac r*, *case-tac r*, *rename-tac m a*, *simp*)
by *simp*

have {*p* :: ('*i*,'*a*) *tagged-packet-scheme*. *?γ*,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} =
{*p* :: ('*i*,'*a*) *tagged-packet-scheme*. *?γ*,*p*⊢ ⟨*?rs1*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
apply(*subst approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF s1*]])
apply(*subst approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers*]])
apply(*rule Collect-cong*)
subgoal for *p* **using** *packet-assume-new*[**where** *p* = *p*] *newpkt-def*[**where** *p* = *p*] **by** *auto*
done
also have {*p*. *?γ*,*p*⊢ ⟨*?rs1*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} =
{*p*. *?γ*,*p*⊢ ⟨*?rs2*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
apply(*subst transform-upper-closure*(*1*)[*OF s1*])
by *simp*
also have ... = {*p*. *?γ*,*p*⊢ ⟨*?rs3*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
apply(*cases rtblo*; *simp*; (*subst iface-try-rewrite-rtbl*[*OF s2 nnf2*] | *subst iface-try-rewrite-no-rtbl*[*OF s2 nnf2*]))
using *wf-ipassmt1 wf-ipassmt2 nospoofing wf-in-doubt-allow routing-no-wildcards correct-routing routing-decided* **by** *simp-all*
also have ... = {*p*. *?γ*,*p*⊢ ⟨*?rs4*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
apply(*subst transform-upper-closure*(*1*)[*OF s3*])
by *simp*
finally have *1*: {*p*. *?γ*,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} =
{*p*. *?γ*,*p*⊢ ⟨*?rs4*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} .
from *abstract-primitive-in-doubt-allow-generic*(*2*)[*OF primitive-matcher-generic-common-matcher nnf4 s4*] **have** *2*:
{*p*. *?γ*,*p*⊢ ⟨*?rs4*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} ⊆
{*p*. *?γ*,*p*⊢ ⟨*?rs5*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
by(*auto simp add*: *abstract-for-simple-firewall-def*)
from *abstract-primitive-in-doubt-allow-generic*(*2*)[*OF primitive-matcher-generic-common-matcher nnf5 s5*] **have** *3*:
{*p*. *?γ*,*p*⊢ ⟨*?rs5*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} ⊆
{*p*. *?γ*,*p*⊢ ⟨*?rs6*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
by(*auto simp add*: *abstract-for-simple-firewall-def*)
have *4*: {*p*. *?γ*,*p*⊢ ⟨*?rs6*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} =

$\{p.\ ?\gamma,p\vdash \langle ?rs7,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow \wedge newpkt\ p\}$
apply(*subst transform-upper-closure*(*1*)[*OF s6*])
by *simp*

let *?fw*=λ*rs p. approximating-bigstep-fun ?*γ *p rs Undecided*
have *approximating-rule*: $\bigwedge p.\ ?\gamma,p\vdash \langle ?rs7,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow \longleftrightarrow ?fw\ ?rs7\ p = Decision\ FinalAllow$
using *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF s7*]] **by** *fast*

from *1 2 3 4* **have** $\{p.\ ?\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow \wedge newpkt\ p\} \subseteq$
$\{p.\ ?\gamma,p\vdash \langle ?rs7,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow \wedge newpkt\ p\}$ **by** *blast*

thus $\{p.\ (common\text{-}matcher,\ in\text{-}doubt\text{-}allow),p\vdash \langle rs,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow \wedge newpkt\ p\} \subseteq$
$\{p.\ simple\text{-}fw\ (to\text{-}simple\text{-}firewall\text{-}without\text{-}interfaces\ ipassmt\ rtblo\ rs)\ p = Decision\ FinalAllow \wedge newpkt\ p\}$
apply(*safe*)
subgoal for *p*
unfolding *to-simple-firewall-without-interfaces-def*
using *to-simple-firewall*[*OF simple-fw-preconditions*, **where** $p = p$] *approximating-rule*[**where** $p = p$] **by** *auto*
done

have *common-primitive-match-to-simple-match-nodisc*:
$Some\ sm = common\text{-}primitive\text{-}match\text{-}to\text{-}simple\text{-}match\ m' \Longrightarrow$
$\neg\ has\text{-}disc\ is\text{-}Iiface\ m' \wedge \neg\ has\text{-}disc\ is\text{-}Oiface\ m' \Longrightarrow iiface\ sm = ifaceAny \wedge oiface\ sm = ifaceAny$
if *prems*: *check-simple-fw-preconditions* [*Rule m' a'*]
for m' :: *'i common-primitive match-expr* **and** a' *sm*
using *prems* **proof**(*induction m' arbitrary*: *sm rule*: *common-primitive-match-to-simple-match.induct*)
case *18* **thus** *?case*
by(*simp add*: *check-simple-fw-preconditions-def normalized-protocols-def*)
next
case (*13 m1 m2*) **thus** *?case*

apply(*simp add*: *check-simple-fw-preconditions-def*)
apply(*case-tac common-primitive-match-to-simple-match m1*)
apply(*simp*; *fail*)
apply(*case-tac common-primitive-match-to-simple-match m2*)
apply(*simp*; *fail*)
apply *simp*
apply(*rename-tac a aa*)
apply(*case-tac a*)
apply(*case-tac aa*)

apply(*simp*)
apply(*simp split*: *option.split-asm*)
using *iface-conjunct-ifaceAny normalized-ifaces-def normalized-protocols-def*
by (*metis has-disc-negated.simps*(*4*) *option.inject*)
qed(*simp-all add*: *check-simple-fw-preconditions-def simple-match-any-def*)

have *to-simple-firewall-no-ifaces*: (⋀*m a*. *Rule m a* ∈ *set rs* ⟹ ¬ *has-disc is-Iiface m* ∧ ¬ *has-disc is-Oiface m*) ⟹
∀ *r*∈*set* (*to-simple-firewall rs*). *iiface* (*match-sel r*) = *ifaceAny* ∧ *oiface* (*match-sel r*) = *ifaceAny*
if *pre1*: *check-simple-fw-preconditions rs* **for** *rs* :: *'i common-primitive rule list*
using *pre1* **apply**(*induction rs*)
apply(*simp add*: *to-simple-firewall-simps*; *fail*)
apply *simp*
apply(*subgoal-tac check-simple-fw-preconditions rs*)
prefer *2*
subgoal by(*simp add*: *check-simple-fw-preconditions-def*)
apply(*rename-tac r rs*, *case-tac r*)
apply *simp*
apply(*simp add*: *to-simple-firewall-simps*)
apply(*simp split*: *option.split*)
apply(*intro conjI*)
apply *blast*
apply(*intro allI impI*)
apply(*subgoal-tac* (∀ *m*∈*set* (*to-simple-firewall rs*). *iiface* (*match-sel m*) = *ifaceAny* ∧ *oiface* (*match-sel m*) = *ifaceAny*))
prefer *2*
subgoal by *blast*
apply(*simp*)
apply(*rename-tac m' a' sm*)
apply(*subgoal-tac* ¬ *has-disc is-Iiface m'* ∧ ¬ *has-disc is-Oiface m'*)
prefer *2*
subgoal by *blast*
apply(*subgoal-tac check-simple-fw-preconditions* [*Rule m' a'*])
prefer *2*
subgoal by(*simp add*: *check-simple-fw-preconditions-def*)
apply(*drule common-primitive-match-to-simple-match-nodisc*)
apply(*simp-all*)
done

from *to-simple-firewall-no-ifaces*[*OF simple-fw-preconditions no-interfaces*] **show**

∀ *r* ∈ *set* (*to-simple-firewall-without-interfaces ipassmt rtblo rs*). *iiface* (*match-sel r*) = *ifaceAny* ∧ *oiface* (*match-sel r*) = *ifaceAny*
unfolding *to-simple-firewall-without-interfaces-def*
by(*simp add*: *to-simple-firewall-def simple-fw-preconditions*)

qed

end
theory *Semantics-Embeddings*
imports *Simple-Firewall/SimpleFw-Compliance Matching-Embeddings Semantics Semantics-Ternary/Semantics-Ternary*
begin

42 Semantics Embedding

42.1 Tactic *in-doubt-allow*

lemma *iptables-bigstep-undecided-to-undecided-in-doubt-allow-approx*:
assumes *agree*: *matcher-agree-on-exact-matches* γ β
and *good*: *good-ruleset rs* **and** *semantics*: $\Gamma,\gamma,p\vdash \langle rs, Undecided\rangle \Rightarrow Undecided$
shows $(\beta, in\text{-}doubt\text{-}allow),p\vdash \langle rs, Undecided\rangle \Rightarrow_\alpha Undecided \vee (\beta, in\text{-}doubt\text{-}allow),p\vdash \langle rs, Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow$
proof −
from *semantics good* **show** *?thesis*
proof(*induction rs Undecided Undecided rule*: *iptables-bigstep-induct*)
case *Skip* **thus** *?case* **by**(*auto intro*: *approximating-bigstep.skip*)
next
case *Log* **thus** *?case* **by**(*auto intro*: *approximating-bigstep.empty approximating-bigstep.log approximating-bigstep.nomatch*)
next
case (*Nomatch m a*)
with *not-exact-match-in-doubt-allow-approx-match*[*OF agree*] **have**
$a \neq Log \Longrightarrow a \neq Empty \Longrightarrow a = Accept \wedge Matching\text{-}Ternary.matches\ (\beta, in\text{-}doubt\text{-}allow)\ m\ a\ p \vee \neg\ Matching\text{-}Ternary.matches\ (\beta, in\text{-}doubt\text{-}allow)\ m\ a\ p$
by(*simp add*: *good-ruleset-alt*) *blast*
thus *?case*
by(*cases a*) (*auto intro*: *approximating-bigstep.empty approximating-bigstep.log approximating-bigstep.accept approximating-bigstep.nomatch*)
next
case (*Seq rs rs1 rs2 t*)
from *Seq* **have** *good-ruleset rs1* **and** *good-ruleset rs2* **by**(*simp-all add*: *good-ruleset-append*)
also from *Seq iptables-bigstep-to-undecided* **have** $t = Undecided$ **by** *simp*
ultimately show *?case* **using** *Seq* **by**(*fastforce intro*: *approximating-bigstep.decision Semantics-Ternary.seq'*)
qed(*simp-all add*: *good-ruleset-def*)
qed

lemma *FinalAllow-approximating-in-doubt-allow*:
assumes *agree*: *matcher-agree-on-exact-matches* γ β
and *good*: *good-ruleset rs* **and** *semantics*: $\Gamma,\gamma,p\vdash \langle rs, Undecided\rangle \Rightarrow Decision\ FinalAllow$
shows $(\beta, in\text{-}doubt\text{-}allow),p\vdash \langle rs, Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow$

proof −
from *semantics good* **show** *?thesis*
proof(*induction rs Undecided Decision FinalAllow rule*: *iptables-bigstep-induct*)
case *Allow* **thus** *?case*
by (*auto intro*: *agree approximating-bigstep.accept in-doubt-allow-allows-Accept*)
next
case (*Seq rs rs1 rs2 t*)
from *Seq* **have** *good1*: *good-ruleset rs1* **and** *good2*: *good-ruleset rs2* **by**(*simp-all add*: *good-ruleset-append*)
show *?case*
proof(*cases t*)
case *Decision* **with** *Seq good1 good2* **show** (*β*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow*
by (*auto intro*: *approximating-bigstep.decision approximating-bigstep.seq dest*: *Semantics.decisionD*)
next
case *Undecided*
with *iptables-bigstep-undecided-to-undecided-in-doubt-allow-approx*[*OF agree good1*] *Seq* **have**
(*β*, *in-doubt-allow*),*p*⊢ ⟨*rs1*, *Undecided*⟩ ⇒$_\alpha$ *Undecided* ∨ (*β*, *in-doubt-allow*),*p*⊢ ⟨*rs1*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* **by** *simp*
with *Undecided Seq good1 good2* **show** (*β*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow*
by (*auto intro*: *approximating-bigstep.seq Semantics-Ternary.seq′ approximating-bigstep.decision*)
qed
next
case *Call-result* **thus** *?case* **by**(*simp add*: *good-ruleset-alt*)
qed
qed

corollary *FinalAllows-subseteq-in-doubt-allow*: *matcher-agree-on-exact-matches γ β* ⟹ *good-ruleset rs* ⟹
{*p*. Γ,*γ*,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*} ⊆ {*p*. (*β*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow*}
using *FinalAllow-approximating-in-doubt-allow* **by** (*metis* (*lifting*, *full-types*) *Collect-mono*)

corollary *new-packets-to-simple-firewall-overapproximation*:
defines *preprocess rs* ≡ *upper-closure* (*optimize-matches abstract-for-simple-firewall* (*upper-closure* (*packet-assume-new rs*)))
and *newpkt p* ≡ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*
fixes *p* :: (′*i*::*len*, ′*pkt-ext*) *tagged-packet-scheme*
assumes *matcher-agree-on-exact-matches γ common-matcher* **and** *simple-ruleset rs*
shows {*p*. Γ,*γ*,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow* ∧ *newpkt p*} ⊆ {*p*.

simple-fw (*to-simple-firewall* (*preprocess rs*)) *p* = *Decision FinalAllow* ∧ *newpkt p*}
proof −
from *assms*(*3*) **have** {*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow* ∧ *newpkt p*} ⊆
{*p*. (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
apply(*drule-tac rs*=*rs* **and** Γ=Γ **in** *FinalAllows-subseteq-in-doubt-allow*)
using *simple-imp-good-ruleset assms*(*4*) **apply** *blast*
by *blast*
thus *?thesis* **unfolding** *newpkt-def preprocess-def* **using** *transform-simple-fw-upper*(*2*)[*OF assms*(*4*)] **by** *blast*
qed

lemma *approximating-bigstep-undecided-to-undecided-in-doubt-allow-approx*: *matcher-agree-on-exact-matches* γ β ⟹
good-ruleset rs ⟹
(β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Undecided* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided* ∨ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalDeny*
apply(*rotate-tac 2*)
apply(*induction rs Undecided Undecided rule*: *approximating-bigstep-induct*)
apply(*simp-all*)
apply (*metis iptables-bigstep.skip*)
apply (*metis iptables-bigstep.empty iptables-bigstep.log iptables-bigstep.nomatch*)
apply(*simp split*: *ternaryvalue.split-asm add*: *matches-case-ternaryvalue-tuple*)
apply (*metis in-doubt-allow-allows-Accept iptables-bigstep.nomatch matches-casesE ternaryvalue.distinct*(*1*) *ternaryvalue.distinct*(*5*))
apply(*case-tac a*)
apply(*simp-all*)
apply (*metis iptables-bigstep.drop iptables-bigstep.nomatch*)
apply (*metis iptables-bigstep.log iptables-bigstep.nomatch*)
apply (*metis iptables-bigstep.nomatch iptables-bigstep.reject*)
apply(*simp add*: *good-ruleset-alt*)
apply(*simp add*: *good-ruleset-alt*)
apply(*simp add*: *good-ruleset-alt*)
apply (*metis iptables-bigstep.empty iptables-bigstep.nomatch*)
apply(*simp add*: *good-ruleset-alt*)
apply(*simp add*: *good-ruleset-append*,*clarify*)
by (*metis approximating-bigstep-to-undecided iptables-bigstep.decision iptables-bigstep.seq*)

lemma *FinalDeny-approximating-in-doubt-allow*: *matcher-agree-on-exact-matches* γ β ⟹
good-ruleset rs ⟹
(β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalDeny* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalDeny*
apply(*rotate-tac 2*)
apply(*induction rs Undecided Decision FinalDeny rule*: *approximating-bigstep-induct*)

apply(*simp-all*)
apply (*metis action.distinct*(*1*) *action.distinct*(*5*) *deny not-exact-match-in-doubt-allow-approx-match*)

apply(*simp add*: *good-ruleset-append*, *clarify*)
apply(*case-tac t*)
apply(*simp*)
apply(*drule*(*2*) *approximating-bigstep-undecided-to-undecided-in-doubt-allow-approx*[**where** Γ=Γ])
apply(*erule disjE*)
apply (*metis iptables-bigstep.seq*)
apply (*metis iptables-bigstep.decision iptables-bigstep.seq*)
by (*metis Decision-approximating-bigstep-fun approximating-semantics-imp-fun iptables-bigstep.decision iptables-bigstep.seq*)

corollary *FinalDenys-subseteq-in-doubt-allow*: *matcher-agree-on-exact-matches* γ β $\Longrightarrow$ *good-ruleset rs* $\Longrightarrow$
{*p*. (β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny*} ⊆ {*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalDeny*}
using *FinalDeny-approximating-in-doubt-allow* **by** (*metis* (*lifting*, *full-types*) *Collect-mono*)

If our approximating firewall (the executable version) concludes that we deny a packet, the exact semantic agrees that this packet is definitely denied!

corollary *matcher-agree-on-exact-matches* γ β $\Longrightarrow$ *good-ruleset rs* $\Longrightarrow$
approximating-bigstep-fun (β, *in-doubt-allow*) *p rs Undecided* = (*Decision FinalDeny*) $\Longrightarrow$ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalDeny*
apply(*frule*(*1*) *FinalDeny-approximating-in-doubt-allow*[**where** *p*=*p* **and** Γ=Γ])
apply(*rule approximating-fun-imp-semantics*)
apply (*metis good-imp-wf-ruleset*)
apply(*simp-all*)
done

42.2 Tactic *in-doubt-deny*

lemma *iptables-bigstep-undecided-to-undecided-in-doubt-deny-approx*: *matcher-agree-on-exact-matches* γ β $\Longrightarrow$
good-ruleset rs $\Longrightarrow$
Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided* $\Longrightarrow$
(β, *in-doubt-deny*),*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Undecided* ∨ (β, *in-doubt-deny*),*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny*
apply(*rotate-tac 2*)
apply(*induction rs Undecided Undecided rule*: *iptables-bigstep-induct*)
apply(*simp-all*)
apply (*metis approximating-bigstep.skip*)
apply (*metis approximating-bigstep.empty approximating-bigstep.log approximating-bigstep.nomatch*)
apply(*case-tac a* = *Log*)
apply (*metis approximating-bigstep.log approximating-bigstep.nomatch*)

apply(*case-tac a = Empty*)
apply (*metis approximating-bigstep.empty approximating-bigstep.nomatch*)
apply(*drule-tac a=a* **in** *not-exact-match-in-doubt-deny-approx-match*)
apply(*simp-all*)
apply(*simp add*: *good-ruleset-alt*)
apply *fast*
apply (*metis approximating-bigstep.drop approximating-bigstep.nomatch approximating-bigstep.reject*)
apply(*frule iptables-bigstep-to-undecided*)
apply(*simp*)
apply(*simp add*: *good-ruleset-append*)
apply (*metis* (*opaque-lifting*, *no-types*) *approximating-bigstep.decision Semantics-Ternary.seq'*)
apply(*simp add*: *good-ruleset-def*)
apply(*simp add*: *good-ruleset-def*)
done

lemma *FinalDeny-approximating-in-doubt-deny*: *matcher-agree-on-exact-matches* γ β $\Longrightarrow$
good-ruleset rs $\Longrightarrow$
Γ,γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalDeny* $\Longrightarrow$ (β, *in-doubt-deny*),*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalDeny*
apply(*rotate-tac 2*)
apply(*induction rs Undecided Decision FinalDeny rule*: *iptables-bigstep-induct*)
apply(*simp-all*)
apply (*metis approximating-bigstep.drop approximating-bigstep.reject in-doubt-deny-denies-DropReject*)
apply(*case-tac t*)
apply(*simp-all*)
prefer *2*
apply(*simp add*: *good-ruleset-append*)
apply (*metis approximating-bigstep.decision approximating-bigstep.seq Semantics.decisionD state.inject*)
apply(*simp add*: *good-ruleset-append*, *clarify*)
apply(*drule*(*2*) *iptables-bigstep-undecided-to-undecided-in-doubt-deny-approx*)
apply(*erule disjE*)
apply (*metis approximating-bigstep.seq*)
apply (*metis approximating-bigstep.decision Semantics-Ternary.seq'*)
apply(*simp add*: *good-ruleset-alt*)
done

lemma *approximating-bigstep-undecided-to-undecided-in-doubt-deny-approx*: *matcher-agree-on-exact-matches* γ β $\Longrightarrow$
good-ruleset rs $\Longrightarrow$
(β, *in-doubt-deny*),*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Undecided* $\Longrightarrow$ Γ,γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided* $\vee$ Γ,γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow*

apply(*rotate-tac 2*)
apply(*induction rs Undecided Undecided rule*: *approximating-bigstep-induct*)
apply(*simp-all*)
apply (*metis iptables-bigstep.skip*)
apply (*metis iptables-bigstep.empty iptables-bigstep.log iptables-bigstep.nomatch*)
apply(*simp split*: *ternaryvalue.split-asm add*: *matches-case-ternaryvalue-tuple*)
apply (*metis in-doubt-allow-allows-Accept iptables-bigstep.nomatch matches-casesE ternaryvalue.distinct(1) ternaryvalue.distinct(5)*)
apply(*case-tac a*)
apply(*simp-all*)
apply (*metis iptables-bigstep.accept iptables-bigstep.nomatch*)
apply (*metis iptables-bigstep.log iptables-bigstep.nomatch*)
apply(*simp add*: *good-ruleset-alt*)
apply(*simp add*: *good-ruleset-alt*)
apply(*simp add*: *good-ruleset-alt*)
apply (*metis iptables-bigstep.empty iptables-bigstep.nomatch*)
apply(*simp add*: *good-ruleset-alt*)
apply(*simp add*: *good-ruleset-append*,*clarify*)
by (*metis approximating-bigstep-to-undecided iptables-bigstep.decision iptables-bigstep.seq*)

lemma *FinalAllow-approximating-in-doubt-deny*: *matcher-agree-on-exact-matches* γ β $\Longrightarrow$
good-ruleset rs $\Longrightarrow$
(β, *in-doubt-deny*),*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalAllow* $\Longrightarrow$ Γ,γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow*
apply(*rotate-tac 2*)
apply(*induction rs Undecided Decision FinalAllow rule*: *approximating-bigstep-induct*)
apply(*simp-all*)
apply (*metis action.distinct(1) action.distinct(5) iptables-bigstep.accept not-exact-match-in-doubt-deny-appr*
apply(*simp add*: *good-ruleset-append*, *clarify*)
apply(*case-tac t*)
apply(*simp*)
apply(*drule*(*2*) *approximating-bigstep-undecided-to-undecided-in-doubt-deny-approx*[**where** Γ=Γ])
apply(*erule disjE*)
apply (*metis iptables-bigstep.seq*)
apply (*metis iptables-bigstep.decision iptables-bigstep.seq*)
by (*metis Decision-approximating-bigstep-fun approximating-semantics-imp-fun iptables-bigstep.decision iptables-bigstep.seq*)

corollary *FinalAllows-subseteq-in-doubt-deny*: *matcher-agree-on-exact-matches* γ β $\Longrightarrow$ *good-ruleset rs* $\Longrightarrow$
{*p*. (β, *in-doubt-deny*),*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalAllow*} $\subseteq$ {*p*. Γ,γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow*}
using *FinalAllow-approximating-in-doubt-deny* **by** (*metis* (*lifting*, *full-types*) *Collect-mono*)

corollary *new-packets-to-simple-firewall-underapproximation*:
defines *preprocess rs* ≡ *lower-closure* (*optimize-matches abstract-for-simple-firewall* (*lower-closure* (*packet-assume-new rs*)))
and *newpkt p* ≡ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*
fixes *p* :: ('*i*::*len*, '*pkt-ext*) *tagged-packet-scheme*
assumes *matcher-agree-on-exact-matches* γ *common-matcher* **and** *simple-ruleset rs*
shows {*p*. *simple-fw* (*to-simple-firewall* (*preprocess rs*)) *p* = *Decision FinalAllow* ∧ *newpkt p*} ⊆ {*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow* ∧ *newpkt p*}
proof −
from *assms*(*3*) **have** {*p*. (*common-matcher*, *in-doubt-deny*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *newpkt p*} ⊆
{*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow* ∧ *newpkt p*}
apply(*drule-tac rs*=*rs* **and** Γ=Γ **in** *FinalAllows-subseteq-in-doubt-deny*)
using *simple-imp-good-ruleset assms*(*4*) **apply** *blast*
by *blast*
thus *?thesis* **unfolding** *newpkt-def preprocess-def* **using** *transform-simple-fw-lower*(*2*)[*OF assms*(*4*)] **by** *blast*
qed

42.3 Approximating Closures

theorem *FinalAllowClosure*:
assumes *matcher-agree-on-exact-matches* γ β **and** *good-ruleset rs*
shows {*p*. (β, *in-doubt-deny*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow*} ⊆ {*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*}
and {*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*} ⊆ {*p*. (β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow*}
apply (*metis FinalAllows-subseteq-in-doubt-deny assms*)
by (*metis FinalAllows-subseteq-in-doubt-allow assms*)

theorem *FinalDenyClosure*:
assumes *matcher-agree-on-exact-matches* γ β **and** *good-ruleset rs*
shows {*p*. (β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalDeny*} ⊆ {*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalDeny*}
and {*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalDeny*} ⊆ {*p*. (β, *in-doubt-deny*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalDeny*}
apply (*metis FinalDenys-subseteq-in-doubt-allow assms*)
by (*metis FinalDeny-approximating-in-doubt-deny assms mem-Collect-eq subsetI*)

42.4 Exact Embedding

lemma *LukassLemma*: **assumes** *agree*: *matcher-agree-on-exact-matches* γ β
and *noUnknown*: (∀ *r* ∈ *set rs*. *ternary-ternary-eval* (*map-match-tac* β *p* (*get-match r*)) ≠ *TernaryUnknown*)
and *good*: *good-ruleset rs*
shows (β,α),*p*⊢ ⟨*rs*, *s*⟩ ⇒$_\alpha$ *t* ⟷ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t*

proof −
 { fix t — if we show it for arbitrary t, we can reuse this fact for the other direction.
  assume *a*: $(\beta,\alpha),p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t$
  from *a good agree noUnknown* **have** $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
   proof(*induction rs s t rule*: *approximating-bigstep-induct*)
   qed(*auto intro*: *approximating-bigstep.intros iptables-bigstep.intros dest*: *iptables-bigstepD dest*: *matches-comply-exact simp*: *good-ruleset-append*)
 } note *1=this*
 {
  assume *a*: $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
  obtain *x* **where** *approximating-bigstep-fun* (β,α) *p rs s* $= x$ **by** *simp*
  with *approximating-fun-imp-semantics*[*OF good-imp-wf-ruleset*[*OF good*]] **have** *x*: $(\beta,\alpha),p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha x$ **by** *fast*
  with *1* **have** $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow x$ **by** *simp*
  with *a iptables-bigstep-deterministic* **have** $x = t$ **by** *metis*
  hence $(\beta,\alpha),p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t$ **using** *x* **by** *blast*
 } note *2=this*
 from *1 2* **show** *?thesis* **by** *blast*
qed

For rulesets without *Call*s, the approximating ternary semantics can perfectly simulate the Boolean semantics.

theorem β_{magic}*-approximating-bigstep-iff-iptables-bigstep*:
 assumes $\forall\, r \in$ *set rs*. $\forall\, c$. *get-action r* $\neq$ *Call c*
 shows $((\beta_{magic}\ \gamma),\alpha),p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t \longleftrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
apply(*rule iffI*)
apply(*induction rs s t rule*: *approximating-bigstep-induct*)
 apply(*auto intro*: *iptables-bigstep.intros simp*: β_{magic}*-matching*)[7]
apply(*insert assms*)
apply(*induction rs s t rule*: *iptables-bigstep-induct*)
 apply(*auto intro*: *approximating-bigstep.intros simp*: β_{magic}*-matching*)
done

corollary β_{magic}*-approximating-bigstep-fun-iff-iptables-bigstep*:
 assumes *good-ruleset rs*
 shows *approximating-bigstep-fun* $(\beta_{magic}\ \gamma,\alpha)$ *p rs s* $= t \longleftrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
apply(*subst approximating-semantics-iff-fun-good-ruleset*[*symmetric*])
 using *assms* **apply** *simp*
apply(*subst* β_{magic}*-approximating-bigstep-iff-iptables-bigstep*[**where** $\Gamma=\Gamma$])
 using *assms* **apply** (*simp add*: *good-ruleset-def*)
by *simp*

The function *optimize-primitive-univ* was only applied to the ternary semantics. It is, in fact, also correct for the Boolean semantics, assuming the *common-matcher*.

lemma *Semantics-optimize-primitive-univ-common-matcher*:
 assumes *matcher-agree-on-exact-matches* γ *common-matcher*

shows *Semantics.matches* γ (*optimize-primitive-univ m*) *p* = *Semantics.matches* γ *m p*
proof −
have *65535* = (− *1*::*16 word*)
by *simp*
then have *port-range*: $\bigwedge$*s e port*. *s* = *0* ∧ *e* = *0xFFFF* ⟶ (*port*::*16 word*) ≤ *0xFFFF*
by (*simp only*:) *simp*
from *assms* **show** *?thesis*
apply(*induction m rule*: *optimize-primitive-univ.induct*)
apply(*auto elim*!: *matcher-agree-on-exact-matches-gammaE*
simp add: *port-range match-ifaceAny ipset-from-cidr-0 ctstate-is-UNIV*)
done
qed

end
theory *Iptables-Semantics*
imports *Semantics-Embeddings Semantics-Ternary/Normalized-Matches*
begin

43 Normalizing Rulesets in the Boolean Big Step Semantics

corollary *normalize-rules-dnf-correct-BooleanSemantics*:
assumes *good-ruleset rs*
shows Γ,γ,*p*⊢ ⟨*normalize-rules-dnf rs*, *s*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t*
proof −
from *assms* **have** *assm'*: *good-ruleset* (*normalize-rules-dnf rs*) **by** (*metis good-ruleset-normalize-rules-dnf*)

from *normalize-rules-dnf-correct assms good-imp-wf-ruleset* **have**
∀ β α. *approximating-bigstep-fun* (β,α) *p* (*normalize-rules-dnf rs*) *s* = *approximating-bigstep-fun* (β,α) *p rs s* **by** *fast*
hence
∀ α. *approximating-bigstep-fun* (β_{magic} γ,α) *p* (*normalize-rules-dnf rs*) *s* = *approximating-bigstep-fun* (β_{magic} γ,α) *p rs s* **by** *fast*
with β_{magic}*-approximating-bigstep-fun-iff-iptables-bigstep assms assm'* **show** *?thesis*
by *metis*
qed

end
theory *Code-Interface*
imports
Common-Primitive-toString
IP-Addresses.IP-Address-Parser
../Call-Return-Unfolding
Transform
No-Spoof
../Simple-Firewall/SimpleFw-Compliance

Simple-Firewall.SimpleFw-toString
Simple-Firewall.Service-Matrix
../Semantics-Ternary/Optimizing
../Semantics-Goto
HOL−Library.Code-Target-Numeral
HOL−Library.Code-Bit-Shifts-for-Arithmetic
begin

44 Code Interface

HACK: rewrite quotes such that they are better printable by Isabelle

definition *quote-rewrite* :: *string* ⇒ *string* **where**
quote-rewrite ≡ *map* (λ*c*. *if c* = *char-of-nat 34 then CHR* ''~'' *else c*)

lemma *quote-rewrite* (''*foo*''@[*char-of-nat 34*]) = ''*foo*~'' **by** *eval*

The parser returns the *'i common-primitive ruleset* not as a map but as an association list. This function converts it

definition *map-of-string-ipv4*
:: (*string* × *32 common-primitive rule list*) *list* ⇒ *string* ⇀ *32 common-primitive rule list* **where**
map-of-string-ipv4 rs = *map-of rs*
definition *map-of-string-ipv6*
:: (*string* × *128 common-primitive rule list*) *list* ⇒ *string* ⇀ *128 common-primitive rule list* **where**
map-of-string-ipv6 rs = *map-of rs*
definition *map-of-string*
:: (*string* × *'i common-primitive rule list*) *list* ⇒ *string* ⇀ *'i common-primitive rule list* **where**
map-of-string rs = *map-of rs*

definition *unfold-ruleset-CHAIN-safe* :: *string* ⇒ *action* ⇒ *'i*::*len common-primitive ruleset* ⇒ *'i common-primitive rule list option* **where**
unfold-ruleset-CHAIN-safe = *unfold-optimize-ruleset-CHAIN optimize-primitive-univ*

lemma (*unfold-ruleset-CHAIN-safe chain a rs* = *Some rs'*) ⟹ *simple-ruleset rs'*
by(*simp add*: *Let-def unfold-ruleset-CHAIN-safe-def unfold-optimize-ruleset-CHAIN-def split*: *if-split-asm*)

definition *unfold-ruleset-CHAIN* :: *string* ⇒ *action* ⇒ *'i*::*len common-primitive ruleset* ⇒ *'i common-primitive rule list* **where**
unfold-ruleset-CHAIN chain default-action rs = *the* (*unfold-ruleset-CHAIN-safe chain default-action rs*)

definition *unfold-ruleset-FORWARD* :: *action* ⇒ *'i::len common-primitive ruleset* ⇒ *'i::len common-primitive rule list* **where**
unfold-ruleset-FORWARD = *unfold-ruleset-CHAIN ''FORWARD''*

definition *unfold-ruleset-INPUT* :: *action* ⇒ *'i::len common-primitive ruleset* ⇒ *'i::len common-primitive rule list* **where**
unfold-ruleset-INPUT = *unfold-ruleset-CHAIN ''INPUT''*

definition *unfold-ruleset-OUTPUT* :: *action* ⇒ *'i::len common-primitive ruleset* ⇒ *'i::len common-primitive rule list* **where**
unfold-ruleset-OUTPUT ≡ *unfold-ruleset-CHAIN ''OUTPUT''*

lemma *let fw* = [*''FORWARD''* ↦ []] *in*
unfold-ruleset-FORWARD action.Drop fw
= [*Rule* (*MatchAny* :: *32 common-primitive match-expr*) *action.Drop*] **by** *eval*

definition *nat-to-8word* :: *nat* ⇒ *8 word* **where**
nat-to-8word i ≡ *of-nat i*

definition *nat-to-16word* :: *nat* ⇒ *16 word* **where**
nat-to-16word i ≡ *of-nat i*

definition *integer-to-16word* :: *integer* ⇒ *16 word* **where**
integer-to-16word i ≡ *nat-to-16word* (*nat-of-integer i*)

context
begin
private definition *is-pos-Extra* :: *'i::len common-primitive negation-type* ⇒ *bool* **where**
is-pos-Extra a ≡ (*case a of Pos* (*Extra* -) ⇒ *True* | - ⇒ *False*)
private definition *get-pos-Extra* :: *'i::len common-primitive negation-type* ⇒ *string* **where**
get-pos-Extra a ≡ (*case a of Pos* (*Extra e*) ⇒ *e* | - ⇒ *undefined*)

fun *compress-parsed-extra*
:: *'i::len common-primitive negation-type list* ⇒ *'i common-primitive negation-type list* **where**
compress-parsed-extra [] = [] |
compress-parsed-extra (*a1#a2#as*) = (*if is-pos-Extra a1* ∧ *is-pos-Extra a2*
then compress-parsed-extra (*Pos* (*Extra* (*get-pos-Extra a1*@*'' ''*@*get-pos-Extra a2*))#*as*)
else a1#compress-parsed-extra (*a2#as*)
) |

```
  compress-parsed-extra (a#as) = a#compress-parsed-extra as

 lemma compress-parsed-extra
  (map Pos [Extra ''-m'', (Extra ''recent'' :: 32 common-primitive),
            Extra ''--update'', Extra ''--seconds'', Extra ''60'',
            IIface (Iface ''foobar''),
            Extra ''--name'', Extra ''DEFAULT'', Extra ''--rsource'']) =
   map Pos [Extra ''-m recent --update --seconds 60'',
            IIface (Iface ''foobar''),
            Extra ''--name DEFAULT --rsource''] by eval

 private lemma eval-ternary-And-Unknown-Unkown:
  eval-ternary-And TernaryUnknown (eval-ternary-And TernaryUnknown tv) =
     eval-ternary-And TernaryUnknown tv
  by(cases tv) (simp-all)

 private lemma is-pos-Extra-alist-and:
  is-pos-Extra a ⟹ alist-and (a#as) = MatchAnd (Match (Extra (get-pos-Extra
a))) (alist-and as)
  apply(cases a)
   apply(simp-all add: get-pos-Extra-def is-pos-Extra-def)
  apply(rename-tac e)
  by(case-tac e)(simp-all)

 private lemma compress-parsed-extra-matchexpr-helper:
  ternary-ternary-eval (map-match-tac common-matcher p (alist-and (compress-parsed-extra
as))) =
       ternary-ternary-eval (map-match-tac common-matcher p (alist-and as))
  proof(induction as rule: compress-parsed-extra.induct)
  case 1 thus ?case by(simp)
  next
  case (2 a1 a2) thus ?case
   apply(simp add: is-pos-Extra-alist-and)
   apply(cases a1)
    apply(simp-all add: eval-ternary-And-Unknown-Unkown)
   done
  next
  case 3 thus ?case by(simp)
  qed
```

This lemma justifies that it is okay to fold together the parsed unknown tokens

```
 lemma compress-parsed-extra-matchexpr:
  matches (common-matcher, α) (alist-and (compress-parsed-extra as)) =
     matches (common-matcher, α) (alist-and as)
  apply(simp add: fun-eq-iff)
  apply(intro allI)
  apply(rule matches-iff-apply-f)
  apply(simp add: compress-parsed-extra-matchexpr-helper)
```

done
end

44.1 L4 Ports Parser Helper

context
begin

Replace all matches on ports with the unspecified *0* protocol with the given *primitive-protocol*.

private definition *fill-l4-protocol-raw*
:: *primitive-protocol* ⇒ *'i*::*len common-primitive negation-type list* ⇒ *'i common-primitive negation-type list*
where
fill-l4-protocol-raw protocol ≡ *NegPos-map*
(λ *m*. *case m of Src-Ports* (*L4Ports x pts*) ⇒ *if x* ≠ *0 then undefined else Src-Ports* (*L4Ports protocol pts*)
| *Dst-Ports* (*L4Ports x pts*) ⇒ *if x* ≠ *0 then undefined else Dst-Ports* (*L4Ports protocol pts*)
| *MultiportPorts* (*L4Ports x pts*) ⇒ *if x* ≠ *0 then undefined else MultiportPorts* (*L4Ports protocol pts*)
| *Prot* - ⇒ *undefined* — there should be no more match on the protocol if it was parsed from an iptables-save line
| *m* ⇒ *m*
)

lemma *fill-l4-protocol-raw TCP* [*Neg* (*Dst* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*127*, *0*, *0*, *0*)) *8*)), *Pos* (*Src-Ports* (*L4Ports 0* [(*22*,*22*)]))] =
[*Neg* (*Dst* (*IpAddrNetmask 0x7F000000 8*)), *Pos* (*Src-Ports* (*L4Ports 6* [(*0x16*, *0x16*)]))] **by** *eval*

fun *fill-l4-protocol*
:: *'i*::*len common-primitive negation-type list* ⇒ *'i*::*len common-primitive negation-type list*
where
fill-l4-protocol [] = [] |
fill-l4-protocol (*Pos* (*Prot* (*Proto protocol*)) # *ms*) = *Pos* (*Prot* (*Proto protocol*)) # *fill-l4-protocol-raw protocol ms* |
fill-l4-protocol (*Pos* (*Src-Ports* -) # -) = *undefined* |
fill-l4-protocol (*Pos* (*Dst-Ports* -) # -) = *undefined* |
fill-l4-protocol (*Pos* (*MultiportPorts* -) # -) = *undefined* |
fill-l4-protocol (*Neg* (*Src-Ports* -) # -) = *undefined* |
fill-l4-protocol (*Neg* (*Dst-Ports* -) # -) = *undefined* |
fill-l4-protocol (*Neg* (*MultiportPorts* -) # -) = *undefined* |
fill-l4-protocol (*m* # *ms*) = *m* # *fill-l4-protocol ms*

lemma *fill-l4-protocol* [*Neg* (*Dst* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*127*, *0*, *0*, *0*)) *8*))
, *Neg* (*Prot* (*Proto UDP*))

, *Pos* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (*127*, *0*, *0*, *0*)) *8*))
, *Pos* (*Prot* (*Proto TCP*))
, *Pos* (*Extra* ''*foo*'')
, *Pos* (*Src-Ports* (*L4Ports 0* [(*22*,*22*)]))
, *Neg* (*Extra* ''*Bar*'')] =
[*Neg* (*Dst* (*IpAddrNetmask 0x7F000000 8*))
, *Neg* (*Prot* (*Proto UDP*))
, *Pos* (*Src* (*IpAddrNetmask 0x7F000000 8*))
, *Pos* (*Prot* (*Proto TCP*))
, *Pos* (*Extra* ''*foo*'')
, *Pos* (*Src-Ports* (*L4Ports TCP* [(*0x16*, *0x16*)]))
, *Neg* (*Extra* ''*Bar*'')] **by** *eval*
end

definition *prefix-to-strange-inverse-cisco-mask*:: *nat* ⇒ (*nat* × *nat* × *nat* × *nat*)
where
prefix-to-strange-inverse-cisco-mask n ≡ *dotdecimal-of-ipv4addr* (*Bit-Operations.not* (*mask n* << *32* − *n*))

lemma *prefix-to-strange-inverse-cisco-mask 8* = (*0*, *255*, *255*, *255*) **by** *eval*
lemma *prefix-to-strange-inverse-cisco-mask 16* = (*0*, *0*, *255*, *255*) **by** *eval*
lemma *prefix-to-strange-inverse-cisco-mask 24* = (*0*, *0*, *0*, *255*) **by** *eval*
lemma *prefix-to-strange-inverse-cisco-mask 32* = (*0*, *0*, *0*, *0*) **by** *eval*

end

45 Parser for iptables-save

theory *Parser6*
imports *Code-Interface*
keywords *parse-ip6tables-save* :: *thy-decl*
begin

ML‹(∗*my personal small library*∗)
fun takeWhile p xs = *take-prefix p xs*;

fun dropWhile p xs = *drop-prefix p xs*;

fun dropWhileInclusive p xs = *drop 1* (*dropWhile p xs*)

(∗*split at the predicate, do NOT keep the position where it was split*∗)
fun split-at p xs = (*takeWhile p xs*, *dropWhileInclusive p xs*);

›

ML-val‹
```
split-at (fn x => x <>  ) (raw-explode foo bar)
```
›

46 An SML Parser for iptables-save

Work in Progress

ML‹
```
local
  fun is-start-of-table table s = s = (*^table);
  fun is-end-of-table s = s = COMMIT;

  fun load-file (thy: theory) (path: string list) =
      let val p =  File.full-path (Resources.master-directory thy) (Path.make path);
          val - = loading file ^File.platform-path p |> writeln;
      in
        if
          not (File.exists p) orelse (File.is-dir p)
        then
          raise Fail File not found
        else
          File.read-lines p
      end;

  fun extract-table - [] = []
   | extract-table table (r::rs) = if not (is-start-of-table table r)
                                   then
                                     extract-table table rs
                                   else
                                     takeWhile (fn x => not (is-end-of-table x)) rs;

  fun writenumloaded table-name table = let
      val - = Loaded ^ Int.toString (length table) ^ lines of the ^table-name^ table
|> writeln;
    in table end;

  fun warn-windows-line-endings lines =
    let
      val warn = fn s => if String.isSuffix \r s
                         then
                           writeln WARNING: windows \\r\\n line ending detected
                         else
                           ()
      val - = map warn lines
    in
      lines
```

```
    end;
in
  fun load-table table thy = load-file thy
      #> warn-windows-line-endings
      #> extract-table table
      #> writenumloaded table;
  val load-filter-table = load-table filter;
end;
›

ML‹
(*keep quoted strings as one token*)
local
  fun collapse-quotes [] = []
    | collapse-quotes (\::ss) = let val (quoted, rest) = split-at (fn x => x <> \) ss
in
                                        \ ^ implode quoted^\ :: rest end
    | collapse-quotes (s::ss) = s :: collapse-quotes ss;
in
  val ipt-explode = raw-explode #> collapse-quotes;
end
›
ML-val‹
ipt-explode ad \as das\ boo;
ipt-explode ad \foobar --boo boo;
ipt-explode ent \\\\\ this;
ipt-explode ;
›



ML‹
datatype parsed-action-type = TypeCall | TypeGoto
datatype parsed-match-action = ParsedMatch of term
                             | ParsedNegatedMatch of term
                             | ParsedAction of parsed-action-type * string;

local (*iptables-save parsers*)
  val is-whitespace = Scan.many (fn x => x =  );

  local (*parser for matches*)
    local
      fun extract-int ss = case ss |> implode |> Int.fromString
                                              of SOME i => i
                                               | NONE   => raise Fail unparsable int;

      fun is-iface-char x = Symbol.is-ascii x andalso
```

```
        (Symbol.is-ascii-letter x orelse Symbol.is-ascii-digit x orelse x = +
         orelse x = * orelse x = . orelse x = -)
  in
    fun mk-nat maxval i = if i < 0 orelse i > maxval
          then
           raise Fail(nat (^Int.toString i^) must be between 0 and ^Int.toString
maxval)
          else (HOLogic.mk-number HOLogic.natT i);

    fun ipNetmask-to-hol (ip,len) = @{const IpAddrNetmask (128)} $ mk-ipv6addr
ip $ mk-nat 128 len;
     fun ipRange-to-hol (ip1,ip2) = @{const IpAddrRange (128)} $ mk-ipv6addr
ip1 $ mk-ipv6addr ip2;


    val parser-ip-cidr = parser-ipv6 --| ($$ /) -- (Scan.many1 Symbol.is-ascii-digit
>> extract-int) >> ipNetmask-to-hol;

    val parser-ip-range = parser-ipv6 --| ($$ -) -- parser-ipv6 >> ipRange-to-hol;

     val parser-ip-addr = parser-ipv6 >> (fn ip => @{const IpAddr (128)} $
mk-ipv6addr ip);

     val parser-interface = Scan.many1 is-iface-char >> (implode #> (fn x =>
@{const Iface} $ HOLogic.mk-string x));

    (*TODO: it would be cool to check for a word boundary after all these strings*)
    val parser-protocol = Scan.this-string tcp >> K @{term TCP :: 8 word}
                       || Scan.this-string udp >> K @{term UDP :: 8 word}
                        || (Scan.this-string icmpv6 (*before icmp*) || Scan.this-string
ipv6-icmp)
                            >> K @{term L4-Protocol.IPv6ICMP}
                       || Scan.this-string icmp >> K @{term ICMP :: 8 word}
                       (*Moar Assigned Internet Protocol Numbers below: *)
                       || Scan.this-string esp >> K @{term L4-Protocol.ESP}
                       || Scan.this-string ah >> K @{term L4-Protocol.AH}
                       || Scan.this-string gre >> K @{term L4-Protocol.GRE}

    val parser-ctstate = Scan.this-string NEW >> K @{const CT-New}
                || Scan.this-string ESTABLISHED >> K @{const CT-Established}
                    || Scan.this-string RELATED >> K @{const CT-Related}
                 || Scan.this-string UNTRACKED >> K @{const CT-Untracked}
                    || Scan.this-string INVALID >> K @{const CT-Invalid}

    val parser-tcp-flag = Scan.this-string SYN >> K @{const TCP-SYN}
                       || Scan.this-string ACK >> K @{const TCP-ACK}
                       || Scan.this-string FIN >> K @{const TCP-FIN}
                       || Scan.this-string RST >> K @{const TCP-RST}
                       || Scan.this-string URG >> K @{const TCP-URG}
```

```
                || Scan.this-string PSH >> K @{const TCP-PSH}

    fun parse-comma-separated-list parser = Scan.repeat (parser --| $$ ,) @@@
(parser >> (fn p => [p]))

    local
      val mk-port-single = mk-nat 65535 #> (fn n => @{const nat-to-16word}
$ n)
      val parse-port-raw = Scan.many1 Symbol.is-ascii-digit >> extract-int
      fun port-tuple-warn (p1,p2) =
            if p1 >= p2
            then
                let val -= writeln (WARNING (in ports): ^Int.toString p1^ >=
^Int.toString p2)
              in (p1, p2) end
            else (p1, p2);
    in
      val parser-port-single-tup = (
            (parse-port-raw --| $$ : -- parse-port-raw)
                  >> (port-tuple-warn #> (fn (p1,p2) => (mk-port-single p1,
mk-port-single p2)))
          || (parse-port-raw  >> (fn p => (mk-port-single p, mk-port-single p)))
        ) >> HOLogic.mk-prod
      end
    val parser-port-single-tup-term = parser-port-single-tup
        >> (fn x => @{term L4Ports 0} $ HOLogic.mk-list @{typ 16 word ×
16 word} [x])

    val parser-port-many1-tup = parse-comma-separated-list parser-port-single-tup
        >> (fn x => @{term L4Ports 0} $ HOLogic.mk-list @{typ 16 word ×
16 word} x)

    val parser-ctstate-set = parse-comma-separated-list parser-ctstate
        >> HOLogic.mk-set @{typ ctstate}

    val parser-tcp-flag-set = parse-comma-separated-list parser-tcp-flag
        >> HOLogic.mk-set @{typ tcp-flag}

    val parser-tcp-flags = (parser-tcp-flag-set --| $$  -- parser-tcp-flag-set)
        >> (fn (m,c) => @{const TCP-Flags} $ m $ c)

    val parser-extra = Scan.many1 (fn x => x <>  andalso Symbol.not-eof x)
        >> (implode #> HOLogic.mk-string);
  end;
  val eoo = Scan.ahead ($$  || Scan.one Symbol.is-eof); (*end of option; word
boundary or eof look-ahead*)

  fun parse-cmd-option-generic (d: term -> parsed-match-action) s t (parser:
string list -> (term * string list)) =
```

```
      Scan.finite Symbol.stopper (is-whitespace |-- Scan.this-string s |-- (parser
>> (fn r => d (t $ r))) --| eoo)

   fun parse-cmd-option (s: string) (t: term) (parser: string list -> (term * string
list)) =
          parse-cmd-option-generic ParsedMatch s t parser;

   (*both negated and not negated primitives*)
   fun parse-cmd-option-negated (s: string) (t: term) (parser: string list -> (term
* string list)) =
      parse-cmd-option-generic ParsedNegatedMatch (! ^s) t parser || parse-cmd-option
s t parser;

    fun parse-cmd-option-negated-singleton s t parser = parse-cmd-option-negated
s t parser >> (fn x => [x])

   (*TODO: is the 'Scan.finite Symbol.stopper' correct here?*)
   (*TODO: eoo here?*)
  fun parse-with-module-prefix (module: string) (parser: (string list -> parsed-match-action
* string list)) =
     (Scan.finite Symbol.stopper (is-whitespace |-- Scan.this-string module)) |--
(Scan.repeat parser)
  in

     val parse-ips = parse-cmd-option-negated-singleton -s  @{const Src (128)}
(parser-ip-cidr || parser-ip-addr)
                  || parse-cmd-option-negated-singleton -d  @{const Dst (128)}
(parser-ip-cidr || parser-ip-addr);


   val parse-iprange = parse-with-module-prefix -m iprange
                        (   parse-cmd-option-negated --src-range  @{const Src
(128)} parser-ip-range
                         || parse-cmd-option-negated --dst-range  @{const Dst
(128)} parser-ip-range);

   val parse-iface = parse-cmd-option-negated-singleton -i  @{const IIface (128)}
parser-interface
                  || parse-cmd-option-negated-singleton -o  @{const OIface (128)}
parser-interface;

   (*TODO type is explicit here*)
   val parse-protocol = parse-cmd-option-negated-singleton -p
           @{term (Prot ∘ Proto) :: primitive-protocol ⇒ 128 common-primitive}
parser-protocol; (*negated?*)

   (*-m tcp requires that there is already an -p tcp, iptables checks that for you,
     we assume valid iptables-save (otherwise the kernel would not load it)
     We will fill the protocols in the L4Ports later*)
```

```
    val parse-tcp-options = parse-with-module-prefix -m tcp
                  (    parse-cmd-option-negated --sport  @{const Src-Ports (128)}
parser-port-single-tup-term
                   || parse-cmd-option-negated --dport  @{const Dst-Ports (128)}
parser-port-single-tup-term
                  || parse-cmd-option-negated --tcp-flags  @{const L4-Flags (128)}
parser-tcp-flags);
    val parse-multiports = parse-with-module-prefix -m multiport
                  (    parse-cmd-option-negated --sports  @{const Src-Ports (128)}
parser-port-many1-tup
                   || parse-cmd-option-negated --dports  @{const Dst-Ports (128)}
parser-port-many1-tup
                  || parse-cmd-option-negated --ports  @{const MultiportPorts (32)}
parser-port-many1-tup);
    val parse-udp-options = parse-with-module-prefix -m udp
                  (    parse-cmd-option-negated --sport  @{const Src-Ports (128)}
parser-port-single-tup-term
                   || parse-cmd-option-negated --dport  @{const Dst-Ports (128)}
parser-port-single-tup-term);

    val parse-ctstate = parse-with-module-prefix -m state
                      (parse-cmd-option-negated --state  @{const CT-State (128)}
parser-ctstate-set)
              || parse-with-module-prefix -m conntrack
                    (parse-cmd-option-negated --ctstate  @{const CT-State (128)}
parser-ctstate-set);

     (*TODO: it would be good to fail if there is a ! in the extra; it might be an
unparsed negation*)
    val parse-unknown = (parse-cmd-option  @{const Extra (128)} parser-extra)
>> (fn x => [x]);
  end;


  local (*parser for target/action*)
    fun is-target-char x = Symbol.is-ascii x andalso
        (Symbol.is-ascii-letter x orelse Symbol.is-ascii-digit x orelse x = - orelse x
= - orelse x = ~)

    fun parse-finite-skipwhite parser = Scan.finite Symbol.stopper (is-whitespace
|-- parser);

    val is-icmp-type = fn x => Symbol.is-ascii-letter x orelse x = - orelse x = 6
  in
    val parser-target = Scan.many1 is-target-char >> implode;

    (*parses: -j MY-CUSTOM-CHAIN*)
     (*The -j may not be the end of the line. example: -j LOG --log-prefix
[IPT-DROP]:*)
```

```
    val parse-target-generic : (string list -> parsed-match-action * string list) =
parse-finite-skipwhite
    (Scan.this-string -j |-- (parser-target >> (fn s => ParsedAction (TypeCall,
s))));

   (*parses: REJECT --reject-with type*)
    val parse-target-reject : (string list -> parsed-match-action * string list) =
parse-finite-skipwhite
     (Scan.this-string -j |-- (Scan.this-string REJECT >> (fn s => ParsedAc-
tion (TypeCall, s)))
       --| ((Scan.this-string --reject-with --| Scan.many1 is-icmp-type) ||
Scan.this-string ));


    val parse-target-goto : (string list -> parsed-match-action * string list) =
parse-finite-skipwhite
      (Scan.this-string -g |-- (parser-target >> (fn s => let val - = writeln
(WARNING: goto in `^s^') in ParsedAction (TypeGoto, s) end)));

   val parse-target : (string list -> parsed-match-action * string list) = parse-target-reject
|| parse-target-goto || parse-target-generic;
  end;
in
  (*parses: -A FORWARD*)
  val parse-table-append : (string list -> (string * string list)) = Scan.this-string
-A |-- parser-target --| is-whitespace;

  (*parses: -s 0.31.123.213/88 --foo-bar -j chain --foobar
   First tries to parse a known field, afterwards, it parses something unknown until
a blank space appears
  *)
  val option-parser : (string list -> (parsed-match-action list) * string list) =
      Scan.recover (parse-ips || parse-iprange
                 || parse-iface
                 || parse-protocol
                 || parse-tcp-options || parse-udp-options || parse-multiports
                 || parse-ctstate
                 || parse-target >> (fn x => [x])) (K parse-unknown);


  (*parse-table-append should be called before option-parser, otherwise -A will sim-
ply be an unknown for option-parser*)

  local
    (*:DOS-PROTECT - [0:0]*)
    val custom-chain-decl-parser = ($$ :) |-- parser-target --| Scan.this-string
-  #> fst;
    (*:INPUT ACCEPT [130:12050]*)
    (*TODO: PREROUTING is only valid if we are in the raw table*)
```

```
    val builtin-chain-decl-parser = ($$ :) |--
        (Scan.this-string INPUT || Scan.this-string FORWARD || Scan.this-string
OUTPUT || Scan.this-string PREROUTING) --|
      ($$  ) -- (Scan.this-string ACCEPT || Scan.this-string DROP) --| ($$  )
#> fst;
    val wrap-builtin-chain = (fn (name, policy) => (name, SOME policy));
    val wrap-custom-chain = (fn name => (name, NONE));
  in
    val chain-decl-parser : (string list -> string * string option) =
      Scan.recover (builtin-chain-decl-parser #> wrap-builtin-chain) (K (custom-chain-decl-parser
#> wrap-custom-chain));
  end
end;
›
```

ML‹
```
local
  fun concat [] = []
    | concat (x :: xs) = x @ concat xs;
in
fun Scan-cons-repeat (parser: ('a -> 'b list * 'a)) (s: 'a) : ('b list * 'a) =
    let val (x, rest) = Scan.repeat parser s in (concat x, rest) end;
end
```
›

ML-val‹(*Scan-cons-repeat option-parser*) (*ipt-explode −i lup −j net−fw*)›
ML-val‹(*Scan-cons-repeat option-parser*) (*ipt-explode*)›
ML-val‹(*Scan-cons-repeat option-parser*) (*ipt-explode −i lup foo*)›
ML-val‹(*Scan-cons-repeat option-parser*) (*ipt-explode −m tcp −−dport 22 −−sport
88*)›
ML-val‹(*Scan-cons-repeat option-parser*) (*ipt-explode −j LOG −−log−prefix \Shorewall:INPUT:REJECT:\
−−log−level 6*)›

ML-val‹
```
val (x, rest) = (Scan-cons-repeat option-parser) (ipt-explode −d 0.31.123.213/11.
−−foo-bar \he he\ −f −i eth0+ −s 0.31.123.213/21 moreextra −j foobar −−log);
map (fn p => case p of ParsedMatch t => type-of t | ParsedAction (-,-) =>
dummyT) x;
map (fn p => case p of ParsedMatch t => Pretty.writeln (Syntax.pretty-term
@{context} t) | ParsedAction (-,a) => writeln (action: ^a)) x;
```
›

ML‹
```
local
  fun parse-rule-options (s: string list) : parsed-match-action list = let
      val (parsed, rest) = (case try (Scan.catch (Scan-cons-repeat option-parser))
```

```
s
                                                    of SOME x => x
                                                | NONE   => raise Fail scanning)
      in
      if rest <> []
      then
        raise Fail (Unparsed: '^implode rest^')
      else
        parsed
      end
      handle Fail m => raise Fail (parse-rule-options: ^m^ for rule '^implode s^');

    fun get-target (ps : parsed-match-action list) : (parsed-action-type * string) option
= let
        val actions = List.mapPartial (fn p => case p of ParsedAction a => SOME
a
                                                | -              => NONE) ps
      in case actions of [] => NONE
                       | [action] => SOME action
                       | - => raise Fail there can be at most one target
      end;

    val get-matches : (parsed-match-action list -> term) =
        List.mapPartial (fn p => case p of
                                   ParsedMatch m => SOME (@{const Pos (128 com-
mon-primitive)} $ m)
                                 | ParsedNegatedMatch m => SOME (@{const Neg (128
common-primitive)} $ m)
                                 | ParsedAction - => NONE)
                     #> HOLogic.mk-list @{typ 128 common-primitive negation-type};


    (*returns: (chainname the rule was appended to, target, matches)*)
    fun parse-rule (s: string) : (string * (parsed-action-type * string) option * term)
= let
        val (chainname, rest) =
          (case try (ipt-explode #> Scan.finite Symbol.stopper parse-table-append) s
                                   of SOME x => x
                                 | NONE   => raise Fail (parse-rule: parse-table-append:
^s));
        val parsed = parse-rule-options rest
      in (chainname, get-target parsed, get-matches parsed) end;
in
  (*returns (parsed chain declarations, parsed appended rules*)
  fun rule-type-partition (rs : string list) : ((string * string option) list * (string *
(parsed-action-type * string) option * term) list) =
      let
        val (chain-decl, rules) = List.partition (String.isPrefix :) rs
      in
```

```
    if not (List.all (String.isPrefix -A) rules)
    then
      raise Fail could not partition rules
    else
    let val parsed-chain-decls = (case try (map (ipt-explode #> chain-decl-parser))
chain-decl
                    of SOME x => x
                     | NONE => raise Fail (could not parse chain declarations:
^implode chain-decl));
          val parsed-rules = map parse-rule rules;
              val  - = Parsed ^ Int.toString (length parsed-chain-decls) ^ chain
declarations |> writeln;
          val  - = Parsed ^ Int.toString (length parsed-rules) ^ rules |> writeln;
        in (parsed-chain-decls, parsed-rules) end
    end
  fun get-chain-decls-policy (ls: ((string * string option) list * (string * (parsed-action-type
* string) option * term) list)) = fst ls
  fun get-parsed-rules (ls: ((string * string option) list * (string * (parsed-action-type
* string) option * term) list)) = snd ls
   val filter-chain-decls-names-only :
       ((string * string option) list * (string * (parsed-action-type * string) option
* term) list) ->
         (string list * (string * (parsed-action-type * string) option * term) list) =
(fn (a,b) => (map fst a, b))
end;
›

ML‹(*create a table with the firewall definition*)
structure FirewallTable = Table(type key = string; val ord = Library.string-ord);
type firewall-table = term list FirewallTable.table;

local
  (* Initialize the table. Create a key for every declared chain. *)
  fun FirewallTable-init chain-decls : firewall-table = FirewallTable.empty
           |> fold (fn entry => fn accu => FirewallTable.update-new (entry, [])
accu) chain-decls;

  (* this takes like forever! *)
  (* apply compress-parsed-extra here?*)
  fun hacky-hack t = (*Code-Evaluation.dynamic-value-strict @{context} (@{const
compress-extra} $ t)*)
    @{const alist-and' (128 common-primitive)} $ (@{const fill-l4-protocol (128)}
$ (@{const compress-parsed-extra (128)} $ t))

  fun mk-MatchExpr t = if fastype-of t <> @{typ 128 common-primitive nega-
tion-type list}
                    then
                      raise Fail Type Error
```

```
                    else
                      hacky-hack t;
  fun mk-Rule-help t a = let val r = @{const Rule (128 common-primitive)} $
(mk-MatchExpr t) $ a in
       if fastype-of r <> @{typ 128 common-primitive rule} then raise Fail Type
error in mk-Rule-help
      else r end;

  fun append table chain rule = case FirewallTable.lookup table chain
      of NONE => raise Fail (uninitialized cahin: ^chain)
      | SOME rules => FirewallTable.update (chain, rules@[rule]) table

   fun mk-Rule (tbl: firewall-table) (chain: string, target : (parsed-action-type *
string) option, t : term) =
    if not (FirewallTable.defined tbl chain)
    then
      raise Fail (undefined chain to be appended: ^chain)
    else case target
    of NONE => mk-Rule-help t @{const action.Empty}
     | SOME (TypeCall, ACCEPT) => mk-Rule-help t @{const action.Accept}
     | SOME (TypeCall, DROP) => mk-Rule-help t @{const action.Drop}
     | SOME (TypeCall, REJECT) => mk-Rule-help t @{const action.Reject}
     | SOME (TypeCall, LOG) => mk-Rule-help t @{const action.Log}
     | SOME (TypeCall, RETURN) => mk-Rule-help t @{const action.Return}
     | SOME (TypeCall, custom) => if not (FirewallTable.defined tbl custom)
                                  then
                                    raise Fail (unknown action: ^custom)
                                  else
                                mk-Rule-help t (@{const action.Call} $ HOLogic.mk-string
custom)
     | SOME (TypeGoto, ACCEPT) => raise Fail Unexpected
     | SOME (TypeGoto, DROP) => raise Fail Unexpected
     | SOME (TypeGoto, REJECT) => raise Fail Unexpected
     | SOME (TypeGoto, LOG) => raise Fail Unexpected
     | SOME (TypeGoto, RETURN) => raise Fail Unexpected
     | SOME (TypeGoto, custom) => if not (FirewallTable.defined tbl custom)
                                  then
                                    raise Fail (unknown action: ^custom)
                                  else
                               mk-Rule-help t (@{const action.Goto} $ HOLogic.mk-string
custom);

  (*val init = FirewallTable-init parsed-chain-decls;*)
  (*map type-of (map (mk-Rule init) parsed-rules);*)

in
  local
    fun append-rule (tbl: firewall-table) (chain: string, target : (parsed-action-type *
string) option, t : term) = append tbl chain (mk-Rule tbl (chain, target, t))
```

```
  in
    fun make-firewall-table (parsed-chain-decls : string list, parsed-rules : (string *
(parsed-action-type * string) option * term) list) =
    fold (fn rule => fn accu => append-rule accu rule) parsed-rules (FirewallTable-init
parsed-chain-decls);
  end
end
›
```

ML‹
```
fun mk-Ruleset (tbl: firewall-table) = FirewallTable.dest tbl
    |> map (fn (k,v) => HOLogic.mk-prod (HOLogic.mk-string k, HOLogic.mk-list
@{typ 128 common-primitive rule} v))
    |> HOLogic.mk-list @{typ string × 128 common-primitive rule list}
›
```

ML‹
```
local
  fun default-policy-action-to-term ACCEPT = @{const action.Accept}
    | default-policy-action-to-term DROP = @{const action.Drop}
    | default-policy-action-to-term a = raise Fail (Not a valid default policy '^a^')
in
  (*chain-name * default-policy*)
  fun preparedefault-policies [] = []
    | preparedefault-policies ((chain-name, SOME default-policy)::ls) =
            (chain-name, default-policy-action-to-term default-policy) :: preparede-
fault-policies ls
    | preparedefault-policies ((-, NONE)::ls) = preparedefault-policies ls
end
›
```

ML‹
```
fun trace-timing (printstr : string) (f : 'a -> 'b) (a : 'a) : 'b =
  let val t0 = Time.now(); in
    let val result = f a; in
    let val t1= Time.now(); in
     let val - = writeln(String.concat [printstr^ (, Time.toString(Time.-(t1,t0)),
seconds)]) in
       result
    end end end end;

fun simplify-code ctxt = let val - = writeln unfolding (this may take a while) ... in
       trace-timing Simplified term (Code-Evaluation.dynamic-value-strict ctxt)
    end
```

```
fun certify-term ctxt t = trace-timing Certified term (Thm.cterm-of ctxt) t
```

ML-val‹(∗*Example: putting it all together*∗)
```
fun parse-iptables-save-global thy (file: string list) : term =
    load-filter-table thy file
    |> rule-type-partition
    |> filter-chain-decls-names-only
    |> make-firewall-table
    |> mk-Ruleset
    |> simplify-code @{context}

(∗
val example = parse-iptables-save @{theory} [Parser-Test, data, iptables−save];

Pretty.writeln (Syntax.pretty-term @{context} example);∗)
```

ML‹
```
local
  fun define-const t name lthy = let
      val binding-name = Thm.def-binding name
      val - = writeln (Defining constant ‘ ^ Binding.name-of binding-name ^ ’);
    in
      lthy
        (∗without Proof-Context.set-stmt, there is an ML stack overflow for large
iptables−save dumps∗)
      (∗Debugged by Makarius, Isabelle2016∗)
       |> Proof-Context.set-stmt false  (∗ FIXME workaround context begin oddity
∗)
    |> Local-Theory.define ((name, NoSyn), ((binding-name, [Code.singleton-default-equation-attrib]),
t)) |> #2
    end;

  fun print-default-policies (ps: (string ∗ term) list) = let
      val - = ps |> map (fn (name, -) =>
            if name <> INPUT andalso name <> FORWARD andalso name <>
OUTPUT
            then
               writeln (WARNING: the chain ‘^name^’ is not a built−in chain of
the filter table)
            else ())
      in ps end;

  fun sanity-check-ruleset ctxt t = let
    val check = Code-Evaluation.dynamic-value-strict ctxt (@{const sanity-wf-ruleset
(128 common-primitive)} $ t)
```

```
    in
      if check <> @{term True} then raise ERROR sanity-wf-ruleset failed else t
    end;
in
  fun parse-iptables-save table name path lthy =
    let
      val prepared = path
            |> load-table table (Proof-Context.theory-of lthy)
            |> rule-type-partition
      val firewall = prepared
            |> filter-chain-decls-names-only
            |> make-firewall-table
            |> mk-Ruleset
            (*this may a while*)
          |> simplify-code lthy (*was: @{context} (*TODO: is this correct here?*)*)
           |> trace-timing checked sanity with sanity-wf-ruleset (sanity-check-ruleset
lthy)
      val default-policis = prepared
            |> get-chain-decls-policy
            |> preparedefault-policies
            |> print-default-policies
    in
      lthy
      |> define-const firewall name
      |> fold (fn (chain-name, policy) =>
         define-const policy (Binding.name (Binding.name-of name ^- ^chain-name
^ -default-policy)))
          default-policis
    end
end
›


ML‹
  Outer-Syntax.local-theory @{command-keyword parse-ip6tables-save}
  load a file generated by iptables−save and make the firewall definition available as
isabelle term
    (Parse.binding --| @{keyword =} -- Scan.repeat1 Parse.path >>
      (fn (binding, paths) => parse-iptables-save filter binding paths))
›


end
theory No-Spoof-Embeddings
imports Semantics-Embeddings
        Primitive-Matchers/No-Spoof
begin
```

47 Spoofing protection in Ternary Semantics implies Spoofing protection Boolean Semantics

If *no-spoofing* is shown in the ternary semantics, it implies that no spoofing is possible in the Boolean semantics with magic oracle. We only assume that the oracle agrees with the *common-matcher* on the not-unknown parts.

lemma *approximating-imp-booloan-semantics-nospoofing*:
assumes *matcher-agree-on-exact-matches* γ *common-matcher*
and *simple-ruleset rs*
and *no-spoofing*: *no-spoofing TYPE('pkt-ext) ipassmt rs*
shows $\forall$ *iface* $\in$ *dom ipassmt*. $\forall$ *p*::('*i*::*len*,'*pkt-ext*) *tagged-packet-scheme*.
(Γ,γ,*p*(|*p-iiface*:=*iface-sel iface*|)⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*)
⟶
p-src p ∈ (*ipcidr-union-set* (*set* (*the* (*ipassmt iface*))))
unfolding *no-spoofing-def*
proof(*intro ballI allI impI*)
fix *iface p*
assume *i*: *iface* ∈ *dom ipassmt*
and *a*: Γ,γ,*p*(|*p-iiface* := *iface-sel iface*|)⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*

from *no-spoofing*[*unfolded no-spoofing-def*] *i* **have** *no-spoofing'*:
(*common-matcher*, *in-doubt-allow*),*p*(|*p-iiface* := *iface-sel iface*|)⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ⟶
p-src p ∈ *ipcidr-union-set* (*set* (*the* (*ipassmt iface*))) **by** *blast*

from *assms simple-imp-good-ruleset FinalAllows-subseteq-in-doubt-allow*[**where** *rs*=*rs*] **have**
{*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*} ⊆ {*p*. (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow*}
by *blast*
with *a* **have** (*common-matcher*, *in-doubt-allow*),*p*(|*p-iiface* := *iface-sel iface*|)⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* **by** *blast*
with *no-spoofing'* **show** *p-src p* ∈ *ipcidr-union-set* (*set* (*the* (*ipassmt iface*)))**by** *blast*
qed

corollary
assumes *matcher-agree-on-exact-matches* γ *common-matcher* **and** *simple-ruleset rs*
and *no-spoofing*: *no-spoofing TYPE('pkt-ext) ipassmt rs* **and** *iface* ∈ *dom ipassmt*
shows {*p-src p* | *p* :: ('*i*::*len*,'*pkt-ext*) *tagged-packet-scheme*. (Γ,γ,*p*(|*p-iiface*:=*iface-sel iface*|)⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*)} ⊆
ipcidr-union-set (*set* (*the* (*ipassmt iface*)))
using *approximating-imp-booloan-semantics-nospoofing*[*OF assms*(*1*) *assms*(*2*) *assms*(*3*), **where** Γ=Γ]

using *assms(4)* **by** *blast*

corollary *no-spoofing-executable-set*:
assumes *matcher-agree-on-exact-matches* γ *common-matcher*
and *simple-ruleset rs*
and $\forall$ *r*$\in$*set rs. normalized-nnf-match* (*get-match r*)
and *no-spoofing-executable*: $\forall$ *iface* $\in$ *dom ipassmt. no-spoofing-iface iface ipassmt rs*
and *iface* $\in$ *dom ipassmt*
shows {*p-src p* | *p* :: (*'i*::*len*,*'pkt-ext*) *tagged-packet-scheme.* (Γ,γ,*p*(|*p-iiface*:=*iface-sel iface*|)$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow*)} $\subseteq$
ipcidr-union-set (*set* (*the* (*ipassmt iface*)))
proof −
{ assume *no-spoofing*: *no-spoofing TYPE*(*'pkt-ext*) *ipassmt rs*
have {*p-src p* | *p* :: (*'i*,*'pkt-ext*) *tagged-packet-scheme.* (Γ,γ,*p*(|*p-iiface*:=*iface-sel iface*|)$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow*)} $\subseteq$
ipcidr-union-set (*set* (*the* (*ipassmt iface*)))
using *approximating-imp-booloan-semantics-nospoofing*[*OF assms(1) assms(2) no-spoofing*, **where** Γ=Γ]
using *assms(5)* **by** *blast*
}
with *no-spoofing-iface*[*OF assms(2) assms(3) no-spoofing-executable*] **show** *?thesis* **by** *blast*
qed

corollary *no-spoofing-executable-set-preprocessed*:
fixes *ipassmt* :: *'i*::*len ipassignment*
defines *preprocess rs* $\equiv$ *upper-closure* (*packet-assume-new rs*)
and *newpkt p* $\equiv$ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) $\wedge$ *p-tag-ctstate p* = *CT-New*
assumes *matcher-agree-on-exact-matches* γ *common-matcher*
and *simplers*: *simple-ruleset rs*
and *no-spoofing-executable*: $\forall$ *iface* $\in$ *dom ipassmt. no-spoofing-iface iface ipassmt* (*preprocess rs*)
and *iface* $\in$ *dom ipassmt*
shows {*p-src p* | *p* :: (*'i*::*len*,*'pkt-ext*) *tagged-packet-scheme. newpkt p* $\wedge$ Γ,γ,*p*(|*p-iiface*:=*iface-sel iface*|)$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow*} $\subseteq$
ipcidr-union-set (*set* (*the* (*ipassmt iface*)))
proof −
have *newpktD*: *newpkt p* $\Longrightarrow$ *newpkt* (*p*(|*p-iiface* := *iface-sel iface*|)) **for** *p*
by(*simp add*: *newpkt-def*)
from *packet-assume-new-simple-ruleset*[*OF simplers*] **have** *s1*: *simple-ruleset* (*packet-assume-new rs*) **.**
from *transform-upper-closure*(*2*)[*OF s1*] **have** *s2*: *simple-ruleset* (*upper-closure* (*packet-assume-new rs*)) **.**
hence *s2'*: *simple-ruleset* (*preprocess rs*) **unfolding** *preprocess-def* **by** *simp*

have $\forall r{\in}set\ (preprocess\ rs).\ normalized\text{-}nnf\text{-}match\ (get\text{-}match\ r)$
unfolding *preprocess-def*
using *transform-upper-closure*(*3*)[*OF s1*] **by** *simp*

from *no-spoofing-iface*[*OF s2' this no-spoofing-executable*] **have** *nospoof*: *no-spoofing TYPE*('*a*) *ipassmt* (*preprocess rs*) **.**

from *assms*(*3*) **have** *1*: $\{p.\ \Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Decision\ FinalAllow \wedge newpkt\ p\} \subseteq$
$\{p.\ (common\text{-}matcher,\ in\text{-}doubt\text{-}allow),p\vdash \langle rs,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow \wedge newpkt\ p\}$
apply(*drule-tac rs*=*rs* **and** Γ=Γ **in** *FinalAllows-subseteq-in-doubt-allow*)
using *simple-imp-good-ruleset assms*(*4*) **apply** *blast*
by *blast*
have *2*: $\{p.\ (common\text{-}matcher,\ in\text{-}doubt\text{-}allow),p\vdash \langle rs,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow \wedge newpkt\ p\} \subseteq$
$\{p.\ (common\text{-}matcher,\ in\text{-}doubt\text{-}allow),p\vdash \langle preprocess\ rs,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow \wedge newpkt\ p\}$
unfolding *newpkt-def preprocess-def*
apply(*subst transform-upper-closure*(*1*)[*OF s1*])
apply(*subst approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF s1*]])
apply(*subst approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers*]])
using *packet-assume-new newpkt-def* **by** *force*
from *1 2* **have** $\{p.\ \Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Decision\ FinalAllow \wedge newpkt\ p\} \subseteq$
$\{p.\ (common\text{-}matcher,\ in\text{-}doubt\text{-}allow),p\vdash \langle preprocess\ rs,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow \wedge newpkt\ p\}$ **by** *simp*
hence $p\colon \Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Decision\ FinalAllow \wedge newpkt\ p \Longrightarrow$
$(common\text{-}matcher,\ in\text{-}doubt\text{-}allow),p\vdash \langle preprocess\ rs,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow \wedge newpkt\ p$ **for** *p* **by** *blast*
have *x*: $\{p\text{-}src\ p \mid p\ .\ newpkt\ p \wedge (\Gamma,\gamma,p(\!|p\text{-}iiface{:=}iface\text{-}sel\ iface|\!)\vdash \langle rs,\ Undecided\rangle \Rightarrow Decision\ FinalAllow)\} \subseteq$
$\{p\text{-}src\ p \mid p\ .\ newpkt\ p \wedge (common\text{-}matcher,\ in\text{-}doubt\text{-}allow),p(\!|p\text{-}iiface{:=}iface\text{-}sel\ iface|\!)\vdash \langle preprocess\ rs,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow\}$
apply(*safe*, *rename-tac p*)
apply(*drule newpktD*)
apply(*rule-tac x*=*p*(|*p-iiface* := *iface-sel iface*|) **in** *exI*)
using *p* **by** *simp*
note[[*show-types*]]
with *nospoof* **have** *y*:
$\{p\text{-}src\ p \mid p :: ('i{::}len,'pkt\text{-}ext)\ tagged\text{-}packet\text{-}scheme.\ newpkt\ p \wedge (common\text{-}matcher, in\text{-}doubt\text{-}allow),p(\!|p\text{-}iiface{:=}iface\text{-}sel\ iface|\!)\vdash \langle preprocess\ rs,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow\}$
$\subseteq ipcidr\text{-}union\text{-}set\ (set\ (the\ (ipassmt\ iface)))$
apply(*simp add*: *no-spoofing-def*)
by(*blast dest*: *bspec*[*OF - assms*(*6*)])
from *x y* **show** *?thesis* **by** *simp*

qed

end

48 Parser for iptables-save

theory *Parser*
imports *Code-Interface*
keywords *parse-iptables-save* :: *thy-decl*
begin

ML‹(∗*my personal small library*∗)
fun takeWhile p xs = take-prefix p xs;

fun dropWhile p xs = drop-prefix p xs;

fun dropWhileInclusive p xs = drop 1 (dropWhile p xs)

(∗*split at the predicate, do NOT keep the position where it was split*∗)
fun split-at p xs = (takeWhile p xs, dropWhileInclusive p xs);
›

ML-val‹
split-at (fn x => x <>) (raw-explode foo bar)
›

49 An SML Parser for iptables-save

Work in Progress

ML‹
local
fun is-start-of-table table s = s = (∗^table);
fun is-end-of-table s = s = COMMIT;

fun load-file (thy: theory) (path: string list) =
let val p = File.full-path (Resources.master-directory thy) (Path.make path);
val - = loading file ^File.platform-path p |> writeln;
in
if
not (File.exists p) orelse (File.is-dir p)
then
raise Fail File not found
else
File.read-lines p
end;

```
  fun extract-table - [] = []
    | extract-table table (r::rs) = if not (is-start-of-table table r)
                                    then
                                      extract-table table rs
                                    else
                                      takeWhile (fn x => not (is-end-of-table x)) rs;

  fun writenumloaded table-name table = let
      val - = Loaded ^ Int.toString (length table) ^ lines of the ^table-name^ table
|> writeln;
    in table end;

  fun warn-windows-line-endings lines =
    let
      val warn = fn s => if String.isSuffix \r s
                         then
                           writeln WARNING: windows \\r\\n line ending detected
                         else
                           ()
      val - = map warn lines
    in
      lines
    end;
in
  fun load-table table thy = load-file thy
      #> warn-windows-line-endings
      #> extract-table table
      #> writenumloaded table;
  val load-filter-table = load-table filter;
end;
›
```

ML‹
```
(*keep quoted strings as one token*)
local
  fun collapse-quotes [] = []
    | collapse-quotes (\::ss) = let val (quoted, rest) = split-at (fn x => x <> \) ss
in
                                      \ ^ implode quoted^\ :: rest end
    | collapse-quotes (s::ss) = s :: collapse-quotes ss;
in
  val ipt-explode = raw-explode #> collapse-quotes;
end
```
›
ML-val‹
```
ipt-explode ad \as das\ boo;
ipt-explode ad \foobar --boo boo;
ipt-explode ent \\\\\ this;
```

```
ipt-explode ;
›

ML‹
datatype parsed-action-type = TypeCall | TypeGoto
datatype parsed-match-action = ParsedMatch of term
                             | ParsedNegatedMatch of term
                             | ParsedAction of parsed-action-type * string;

local (*iptables−save parsers*)
  val is-whitespace = Scan.many (fn x => x =  );

  local (*parser for matches*)
    local
      fun extract-int ss = case ss |> implode |> Int.fromString
                                                  of SOME i => i
                                                  | NONE   => raise Fail unparsable int;

      fun is-iface-char x = Symbol.is-ascii x andalso
            (Symbol.is-ascii-letter x orelse Symbol.is-ascii-digit x orelse x = +
             orelse x = * orelse x = . orelse x = −)
    in
      fun mk-nat maxval i = if i < 0 orelse i > maxval
                then
                 raise Fail(nat (^Int.toString i^) must be between 0 and ^Int.toString
maxval)
                else (HOLogic.mk-number HOLogic.natT i);

     fun ipNetmask-to-hol (ip,len) = @{const IpAddrNetmask (32)} $ mk-ipv4addr
ip $ mk-nat 32 len;
      fun ipRange-to-hol (ip1,ip2) = @{const IpAddrRange (32)} $ mk-ipv4addr
ip1 $ mk-ipv4addr ip2;


    val parser-ip-cidr = parser-ipv4 −−| ($$ /) −− (Scan.many1 Symbol.is-ascii-digit
>> extract-int) >> ipNetmask-to-hol;

    val parser-ip-range = parser-ipv4 −−| ($$ −) −− parser-ipv4 >> ipRange-to-hol;

       val parser-ip-addr = parser-ipv4 >> (fn ip => @{const IpAddr (32)} $
mk-ipv4addr ip);

      val parser-interface = Scan.many1 is-iface-char >> (implode #> (fn x =>
@{const Iface} $ HOLogic.mk-string x));

     (*TODO: it would be cool to check for a word boundary after all these strings*)
```

```
val parser-protocol = Scan.this-string tcp >> K @{term TCP :: 8 word}
                    || Scan.this-string udp >> K @{term UDP :: 8 word}
                     || (Scan.this-string icmpv6 (*before icmp*) || Scan.this-string
ipv6-icmp)
                         >> K @{term L4-Protocol.IPv6ICMP}
                    || Scan.this-string icmp >> K @{term ICMP :: 8 word}
                    (*Moar Assigned Internet Protocol Numbers below: *)
                    || Scan.this-string esp >> K @{term L4-Protocol.ESP}
                    || Scan.this-string ah >> K @{term L4-Protocol.AH}
                    || Scan.this-string gre >> K @{term L4-Protocol.GRE}

val parser-ctstate = Scan.this-string NEW >> K @{const CT-New}
                || Scan.this-string ESTABLISHED >> K @{const CT-Established}
                    || Scan.this-string RELATED >> K @{const CT-Related}
                  || Scan.this-string UNTRACKED >> K @{const CT-Untracked}
                    || Scan.this-string INVALID >> K @{const CT-Invalid}

val parser-tcp-flag = Scan.this-string SYN >> K @{const TCP-SYN}
                    || Scan.this-string ACK >> K @{const TCP-ACK}
                    || Scan.this-string FIN >> K @{const TCP-FIN}
                    || Scan.this-string RST >> K @{const TCP-RST}
                    || Scan.this-string URG >> K @{const TCP-URG}
                    || Scan.this-string PSH >> K @{const TCP-PSH}

fun parse-comma-separated-list parser = Scan.repeat (parser --| $$ ,) @@@
(parser >> (fn p => [p]))

local
  val mk-port-single = mk-nat 65535 #> (fn n => @{const nat-to-16word}
$ n)
  val parse-port-raw = Scan.many1 Symbol.is-ascii-digit >> extract-int
  fun port-tuple-warn (p1,p2) =
        if p1 >= p2
        then
            let val -= writeln (WARNING (in ports): ^Int.toString p1^ >=
^Int.toString p2)
              in (p1, p2) end
        else (p1, p2);
in
  val parser-port-single-tup = (
        (parse-port-raw --| $$ : -- parse-port-raw)
              >> (port-tuple-warn #> (fn (p1,p2) => (mk-port-single p1,
mk-port-single p2)))
       || (parse-port-raw  >> (fn p => (mk-port-single p, mk-port-single p)))
     ) >> HOLogic.mk-prod
  end
val parser-port-single-tup-term = parser-port-single-tup
      >> (fn x => @{term L4Ports 0} $ HOLogic.mk-list @{typ 16 word ×
16 word} [x])
```

```
    val parser-port-many1-tup = parse-comma-separated-list parser-port-single-tup
          >> (fn x => @{term L4Ports 0} $ HOLogic.mk-list @{typ 16 word ×
16 word} x)

    val parser-ctstate-set = parse-comma-separated-list parser-ctstate
          >> HOLogic.mk-set @{typ ctstate}

    val parser-tcp-flag-set = parse-comma-separated-list parser-tcp-flag
          >> HOLogic.mk-set @{typ tcp-flag}

    val parser-tcp-flags = (parser-tcp-flag-set --| $$   -- parser-tcp-flag-set)
          >> (fn (m,c) => @{const TCP-Flags} $ m $ c)

    val parser-extra = Scan.many1 (fn x => x <>   andalso Symbol.not-eof x)
          >> (implode #> HOLogic.mk-string);
  end;
   val eoo = Scan.ahead ($$   || Scan.one Symbol.is-eof); (*end of option; word
boundary or eof look-ahead*)

   fun parse-cmd-option-generic (d: term -> parsed-match-action) s t (parser:
string list -> (term * string list)) =
     Scan.finite Symbol.stopper (is-whitespace |-- Scan.this-string s |-- (parser
>> (fn r => d (t $ r))) --| eoo)

  fun parse-cmd-option (s: string) (t: term) (parser: string list -> (term * string
list)) =
        parse-cmd-option-generic ParsedMatch s t parser;

   (*both negated and not negated primitives*)
   fun parse-cmd-option-negated (s: string) (t: term) (parser: string list -> (term
* string list)) =
     parse-cmd-option-generic ParsedNegatedMatch (! ^s) t parser || parse-cmd-option
s t parser;

   fun parse-cmd-option-negated-singleton s t parser = parse-cmd-option-negated
s t parser >> (fn x => [x])

   (*TODO: is the 'Scan.finite Symbol.stopper' correct here?*)
   (*TODO: eoo here?*)
  fun parse-with-module-prefix (module: string) (parser: (string list -> parsed-match-action
* string list)) =
    (Scan.finite Symbol.stopper (is-whitespace |-- Scan.this-string module)) |--
(Scan.repeat parser)
  in

    val parse-ips = parse-cmd-option-negated-singleton -s   @{const Src (32)}
(parser-ip-cidr || parser-ip-addr)
                  || parse-cmd-option-negated-singleton -d   @{const Dst (32)}
```

```
(parser-ip-cidr || parser-ip-addr);


    val parse-iprange = parse-with-module-prefix -m iprange
                            (    parse-cmd-option-negated --src-range  @{const Src
(32)} parser-ip-range
                             || parse-cmd-option-negated --dst-range  @{const Dst
(32)} parser-ip-range);

    val parse-iface = parse-cmd-option-negated-singleton -i  @{const IIface (32)}
parser-interface
                      || parse-cmd-option-negated-singleton -o  @{const OIface (32)}
parser-interface;

    (*TODO type is explicit here*)
    val parse-protocol = parse-cmd-option-negated-singleton -p
              @{term (Prot ∘ Proto) :: primitive-protocol ⇒ 32 common-primitive}
parser-protocol; (*negated?*)

    (*-m tcp requires that there is already an -p tcp, iptables checks that for you,
      we assume valid iptables-save (otherwise the kernel would not load it)
      We will fill the protocols in the L4Ports later*)
    val parse-tcp-options = parse-with-module-prefix -m tcp
                  (     parse-cmd-option-negated --sport   @{const Src-Ports (32)}
parser-port-single-tup-term
                    || parse-cmd-option-negated --dport   @{const Dst-Ports (32)}
parser-port-single-tup-term
                  || parse-cmd-option-negated --tcp-flags  @{const L4-Flags (32)}
parser-tcp-flags);
    val parse-multiports = parse-with-module-prefix -m multiport
                  (     parse-cmd-option-negated --sports  @{const Src-Ports (32)}
parser-port-many1-tup
                    || parse-cmd-option-negated --dports  @{const Dst-Ports (32)}
parser-port-many1-tup
                || parse-cmd-option-negated --ports  @{const MultiportPorts (32)}
parser-port-many1-tup);
    val parse-udp-options = parse-with-module-prefix -m udp
                  (     parse-cmd-option-negated --sport   @{const Src-Ports (32)}
parser-port-single-tup-term
                    || parse-cmd-option-negated --dport   @{const Dst-Ports (32)}
parser-port-single-tup-term);

    val parse-ctstate = parse-with-module-prefix -m state
                       (parse-cmd-option-negated --state   @{const CT-State (32)}
parser-ctstate-set)
              || parse-with-module-prefix -m conntrack
                     (parse-cmd-option-negated --ctstate   @{const CT-State (32)}
parser-ctstate-set);
```

```
    (*TODO: it would be good to fail if there is a ! in the extra; it might be an unparsed negation*)
   val parse-unknown = (parse-cmd-option @{const Extra (32)} parser-extra) >> (fn x => [x]);
  end;


  local (*parser for target/action*)
    fun is-target-char x = Symbol.is-ascii x andalso
        (Symbol.is-ascii-letter x orelse Symbol.is-ascii-digit x orelse x = - orelse x = - orelse x = ~)

     fun parse-finite-skipwhite parser = Scan.finite Symbol.stopper (is-whitespace |-- parser);

    val is-icmp-type = fn x => Symbol.is-ascii-letter x orelse x = - orelse x = 6
  in
    val parser-target = Scan.many1 is-target-char >> implode;

    (*parses: -j MY-CUSTOM-CHAIN*)
     (*The -j may not be the end of the line. example: -j LOG --log-prefix [IPT-DROP]:*)
     val parse-target-generic : (string list -> parsed-match-action * string list) = parse-finite-skipwhite
     (Scan.this-string -j |-- (parser-target >> (fn s => ParsedAction (TypeCall, s))));

    (*parses: REJECT --reject-with type*)
     val parse-target-reject : (string list -> parsed-match-action * string list) = parse-finite-skipwhite
     (Scan.this-string -j |-- (Scan.this-string REJECT >> (fn s => ParsedAction (TypeCall, s)))
        --| ((Scan.this-string --reject-with --| Scan.many1 is-icmp-type) || Scan.this-string ));


     val parse-target-goto : (string list -> parsed-match-action * string list) = parse-finite-skipwhite
       (Scan.this-string -g |-- (parser-target >> (fn s => let val - = writeln (WARNING: goto in `^s^`) in ParsedAction (TypeGoto, s) end)));

   val parse-target : (string list -> parsed-match-action * string list) = parse-target-reject || parse-target-goto || parse-target-generic;
  end;
in
  (*parses: -A FORWARD*)
  val parse-table-append : (string list -> (string * string list)) = Scan.this-string -A |-- parser-target --| is-whitespace;
```

```
(*parses: -s 0.31.123.213/88 --foo-bar -j chain --foobar
 First tries to parse a known field, afterwards, it parses something unknown until
a blank space appears
*)
val option-parser : (string list -> (parsed-match-action list) * string list) =
    Scan.recover (parse-ips || parse-iprange
                || parse-iface
                || parse-protocol
                || parse-tcp-options || parse-udp-options || parse-multiports
                || parse-ctstate
                || parse-target >> (fn x => [x])) (K parse-unknown);


(*parse-table-append should be called before option-parser, otherwise -A will sim-
ply be an unknown for option-parser*)

local
  (*:DOS-PROTECT - [0:0]*)
   val custom-chain-decl-parser = ($$ :) |-- parser-target --| Scan.this-string
-  #> fst;
  (*:INPUT ACCEPT [130:12050]*)
  (*TODO: PREROUTING is only valid if we are in the raw table*)
  val builtin-chain-decl-parser = ($$ :) |--
     (Scan.this-string INPUT || Scan.this-string FORWARD || Scan.this-string
OUTPUT || Scan.this-string PREROUTING) --|
    ($$  ) -- (Scan.this-string ACCEPT || Scan.this-string DROP) --| ($$  )
#> fst;
  val wrap-builtin-chain = (fn (name, policy) => (name, SOME policy));
  val wrap-custom-chain = (fn name => (name, NONE));
in
  val chain-decl-parser : (string list -> string * string option) =
     Scan.recover (builtin-chain-decl-parser #> wrap-builtin-chain) (K (custom-chain-decl-parser
#> wrap-custom-chain));
end
end;
›


ML‹
local
  fun concat [] = []
    | concat (x :: xs) = x @ concat xs;
in
fun Scan-cons-repeat (parser: ('a -> 'b list * 'a)) (s: 'a) : ('b list * 'a) =
    let val (x, rest) = Scan.repeat parser s in (concat x, rest) end;
end
›
```

ML-val‹(*Scan-cons-repeat option-parser*) (*ipt-explode −i lup −j net−fw*)›
ML-val‹(*Scan-cons-repeat option-parser*) (*ipt-explode*)›
ML-val‹(*Scan-cons-repeat option-parser*) (*ipt-explode −i lup foo*)›
ML-val‹(*Scan-cons-repeat option-parser*) (*ipt-explode −m tcp −−dport 22 −−sport 88*)›
ML-val‹(*Scan-cons-repeat option-parser*) (*ipt-explode −j LOG −−log−prefix \Shorewall:INPUT:REJECT:\ −−log−level 6*)›

ML-val‹
val (*x*, *rest*) = (*Scan-cons-repeat option-parser*) (*ipt-explode −d 0.31.123.213/11. −−foo-bar \he he\ −f −i eth0+ −s 0.31.123.213/21 moreextra −j foobar −−log*);
map (*fn p => case p of ParsedMatch t => type-of t | ParsedAction* (-,-) => *dummyT*) *x*;
map (*fn p => case p of ParsedMatch t => Pretty.writeln* (*Syntax.pretty-term* @{*context*} *t*) | *ParsedAction* (-,*a*) => *writeln* (*action*: ^*a*)) *x*;
›

ML‹
local
  fun parse-rule-options (*s*: *string list*) : *parsed-match-action list* = *let*
      val (*parsed*, *rest*) = (*case try* (*Scan.catch* (*Scan-cons-repeat option-parser*)) *s*
                                                  of SOME x => x
                                                | *NONE => raise Fail scanning*)
    in
    if rest <> []
    then
      raise Fail (*Unparsed*: ‘^*implode rest*^’)
    else
      parsed
    end
    handle Fail m => raise Fail (*parse-rule-options*: ^*m*^ *for rule* ‘^*implode s*^’);

  fun get-target (*ps* : *parsed-match-action list*) : (*parsed-action-type* ∗ *string*) *option* = *let*
      val actions = List.mapPartial (*fn p => case p of ParsedAction a => SOME a*
                                                | - => *NONE*) *ps*
    in case actions of [] => *NONE*
                | [*action*] => *SOME action*
                | - => *raise Fail there can be at most one target*
    end;

  val get-matches : (*parsed-match-action list −> term*) =
      List.mapPartial (*fn p => case p of*
                ParsedMatch m => SOME (@{*const Pos* (*32 common-primitive*)} $ *m*)
                              | *ParsedNegatedMatch m => SOME* (@{*const Neg* (*32*

```
common-primitive)} $ m)
                        | ParsedAction - => NONE)
                    #> HOLogic.mk-list @{typ 32 common-primitive negation-type};


  (*returns: (chainname the rule was appended to, target, matches)*)
  fun parse-rule (s: string) : (string * (parsed-action-type * string) option * term)
= let
      val (chainname, rest) =
        (case try (ipt-explode #> Scan.finite Symbol.stopper parse-table-append) s
                                of SOME x => x
                             | NONE  => raise Fail (parse-rule: parse-table-append:
^s));
      val parsed = parse-rule-options rest
    in (chainname, get-target parsed, get-matches parsed) end;
in
  (*returns (parsed chain declarations, parsed appended rules*)
  fun rule-type-partition (rs : string list) : ((string * string option) list * (string *
(parsed-action-type * string) option * term) list) =
    let
      val (chain-decl, rules) = List.partition (String.isPrefix :) rs
    in
    if not (List.all (String.isPrefix -A) rules)
    then
      raise Fail could not partition rules
    else
    let val parsed-chain-decls = (case try (map (ipt-explode #> chain-decl-parser))
chain-decl
                    of SOME x => x
                       | NONE => raise Fail (could not parse chain declarations:
^implode chain-decl));
          val parsed-rules = map parse-rule rules;
              val  - = Parsed ^ Int.toString (length parsed-chain-decls) ^ chain
declarations |> writeln;
          val  - = Parsed ^ Int.toString (length parsed-rules) ^ rules |> writeln;
       in (parsed-chain-decls, parsed-rules) end
    end
  fun get-chain-decls-policy (ls: ((string * string option) list * (string * (parsed-action-type
* string) option * term) list)) = fst ls
  fun get-parsed-rules (ls: ((string * string option) list * (string * (parsed-action-type
* string) option * term) list)) = snd ls
   val filter-chain-decls-names-only :
      ((string * string option) list * (string * (parsed-action-type * string) option
* term) list) ->
        (string list * (string * (parsed-action-type * string) option * term) list) =
(fn (a,b) => (map fst a, b))
end;
›
```

```
ML‹(*create a table with the firewall definition*)
structure FirewallTable = Table(type key = string; val ord = Library.string-ord);
type firewall-table = term list FirewallTable.table;

local
  (* Initialize the table. Create a key for every declared chain. *)
  fun FirewallTable-init chain-decls : firewall-table = FirewallTable.empty
            |> fold (fn entry => fn accu => FirewallTable.update-new (entry, [])
accu) chain-decls;

  (* this takes like forever! *)
  (* apply compress-parsed-extra here?*)
  fun hacky-hack t = (*Code-Evaluation.dynamic-value-strict @{context} (@{const
compress-extra} $ t)*)
    @{const alist-and' (32 common-primitive)} $ (@{const fill-l4-protocol (32)} $
(@{const compress-parsed-extra (32)} $ t))

  fun mk-MatchExpr t = if fastype-of t <> @{typ 32 common-primitive nega-
tion-type list}
                       then
                         raise Fail Type Error
                       else
                         hacky-hack t;
  fun mk-Rule-help t a = let val r = @{const Rule (32 common-primitive)} $
(mk-MatchExpr t) $ a in
     if fastype-of r <> @{typ 32 common-primitive rule} then raise Fail Type error
in mk-Rule-help
      else r end;

  fun append table chain rule = case FirewallTable.lookup table chain
     of NONE => raise Fail (uninitialized cahin: ^chain)
     |  SOME rules => FirewallTable.update (chain, rules@[rule]) table

  fun mk-Rule (tbl: firewall-table) (chain: string, target : (parsed-action-type *
string) option, t : term) =
    if not (FirewallTable.defined tbl chain)
    then
      raise Fail (undefined chain to be appended: ^chain)
    else case target
    of NONE => mk-Rule-help t @{const action.Empty}
     | SOME (TypeCall, ACCEPT) => mk-Rule-help t @{const action.Accept}
     | SOME (TypeCall, DROP) => mk-Rule-help t @{const action.Drop}
     | SOME (TypeCall, REJECT) => mk-Rule-help t @{const action.Reject}
     | SOME (TypeCall, LOG) => mk-Rule-help t @{const action.Log}
     | SOME (TypeCall, RETURN) => mk-Rule-help t @{const action.Return}
     | SOME (TypeCall, custom) => if not (FirewallTable.defined tbl custom)
                                  then
                                    raise Fail (unknown action: ^custom)
```

```
                        else
                      mk-Rule-help t (@{const action.Call} $ HOLogic.mk-string
custom)
    | SOME (TypeGoto, ACCEPT) => raise Fail Unexpected
    | SOME (TypeGoto, DROP) => raise Fail Unexpected
    | SOME (TypeGoto, REJECT) => raise Fail Unexpected
    | SOME (TypeGoto, LOG) => raise Fail Unexpected
    | SOME (TypeGoto, RETURN) => raise Fail Unexpected
    | SOME (TypeGoto, custom) => if not (FirewallTable.defined tbl custom)
                        then
                          raise Fail (unknown action: ^custom)
                        else
                      mk-Rule-help t (@{const action.Goto} $ HOLogic.mk-string
custom);

  (*val init = FirewallTable-init parsed-chain-decls;*)
  (*map type-of (map (mk-Rule init) parsed-rules);*)

in
  local
    fun append-rule (tbl: firewall-table) (chain: string, target : (parsed-action-type *
string) option, t : term) = append tbl chain (mk-Rule tbl (chain, target, t))
  in
    fun make-firewall-table (parsed-chain-decls : string list, parsed-rules : (string *
(parsed-action-type * string) option * term) list) =
    fold (fn rule => fn accu => append-rule accu rule) parsed-rules (FirewallTable-init
parsed-chain-decls);
  end
end
```

ML‹
```
fun mk-Ruleset (tbl: firewall-table) = FirewallTable.dest tbl
   |> map (fn (k,v) => HOLogic.mk-prod (HOLogic.mk-string k, HOLogic.mk-list
@{typ 32 common-primitive rule} v))
   |> HOLogic.mk-list @{typ string × 32 common-primitive rule list}
```
›

ML‹
```
local
  fun default-policy-action-to-term ACCEPT = @{const action.Accept}
   | default-policy-action-to-term DROP = @{const action.Drop}
   | default-policy-action-to-term a = raise Fail (Not a valid default policy '^a^')
in
  (*chain-name * default-policy*)
  fun preparedefault-policies [] = []
```

```
  | preparedefault-policies ((chain-name, SOME default-policy)::ls) =
          (chain-name, default-policy-action-to-term default-policy) :: preparede-
fault-policies ls
  | preparedefault-policies ((-, NONE)::ls) = preparedefault-policies ls
end
```

ML‹
```
fun trace-timing (printstr : string) (f : 'a -> 'b) (a : 'a) : 'b =
  let val t0 = Time.now(); in
    let val result = f a; in
    let val t1= Time.now(); in
     let val - = writeln(String.concat [printstr^(, Time.toString(Time.-(t1,t0)),
seconds)]) in
      result
    end end end end;

fun simplify-code ctxt = let val - = writeln unfolding (this may take a while) ... in
      trace-timing Simplified term (Code-Evaluation.dynamic-value-strict ctxt)
    end

fun certify-term ctxt t = trace-timing Certified term (Thm.cterm-of ctxt) t
```

ML-val‹(∗Example: putting it all together∗)
```
fun parse-iptables-save-global thy (file: string list) : term =
    load-filter-table thy file
    |> rule-type-partition
    |> filter-chain-decls-names-only
    |> make-firewall-table
    |> mk-Ruleset
    |> simplify-code @{context}

(∗
val example = parse-iptables-save @{theory} [Parser-Test, data, iptables-save];

Pretty.writeln (Syntax.pretty-term @{context} example);∗)
```

ML‹
```
local
  fun define-const t name lthy = let
      val binding-name = Thm.def-binding name
      val - = writeln (Defining constant ‘ ^ Binding.name-of binding-name ^ ’);
    in
      lthy
```

```
      (*without Proof-Context.set-stmt, there is an ML stack overflow for large
iptables-save dumps*)
     (*Debugged by Makarius, Isabelle2016*)
      |> Proof-Context.set-stmt false  (* FIXME workaround context begin oddity
*)
   |> Local-Theory.define ((name, NoSyn), ((binding-name, [Code.singleton-default-equation-attrib]),
t)) |> #2
   end;

 fun print-default-policies (ps: (string * term) list) = let
     val - = ps |> map (fn (name, -) =>
           if name <> INPUT andalso name <> FORWARD andalso name <>
OUTPUT
           then
              writeln (WARNING: the chain `^name^' is not a built-in chain of
the filter table)
           else ())
     in ps end;

 fun sanity-check-ruleset ctxt t = let
   val check = Code-Evaluation.dynamic-value-strict ctxt (@{const sanity-wf-ruleset
(32 common-primitive)} $ t)
   in
    if check <> @{term True} then raise ERROR sanity-wf-ruleset failed else t
   end;
in
 fun parse-iptables-save table name path lthy =
   let
     val prepared = path
          |> load-table table (Proof-Context.theory-of lthy)
          |> rule-type-partition
     val firewall = prepared
          |> filter-chain-decls-names-only
          |> make-firewall-table
          |> mk-Ruleset
          (*this may a while*)
        |> simplify-code lthy (*was: @{context} (*TODO: is this correct here?*)*)
         |> trace-timing checked sanity with sanity-wf-ruleset (sanity-check-ruleset
lthy)
     val default-policis = prepared
          |> get-chain-decls-policy
          |> preparedefault-policies
          |> print-default-policies
   in
     lthy
     |> define-const firewall name
     |> fold (fn (chain-name, policy) =>
       define-const policy (Binding.name (Binding.name-of name ^- ^chain-name
^ -default-policy)))
```

```
        default-policis
    end
end
›

ML‹
  Outer-Syntax.local-theory @{command-keyword parse-iptables-save}
  load a file generated by iptables−save and make the firewall definition available as
isabelle term
    (Parse.binding −−| @{keyword =} −− Scan.repeat1 Parse.path >>
      (fn (binding, paths) => parse-iptables-save filter binding paths))
›


end
theory Code-haskell
imports
  Routing.IpRoute-Parser
  Primitive-Matchers/Parser
begin

definition word-less-eq :: ('a::len) word ⇒ ('a::len) word ⇒ bool where
  word-less-eq a b ≡ a ≤ b

definition word-to-nat :: ('a::len) word ⇒ nat where
  word-to-nat = Word.unat


definition mk-Set :: 'a list ⇒ 'a set where
  mk-Set = set
```

Assumes that you call *fill-l4-protocol* after parsing!

```
definition mk-L4Ports-pre :: raw-ports ⇒ ipt-l4-ports where
  mk-L4Ports-pre ports-raw = L4Ports 0 ports-raw


fun ipassmt-iprange-translate :: 'i::len ipt-iprange list negation-type ⇒ ('i word ×
nat) list where
  ipassmt-iprange-translate (Pos ips) = concat (map ipt-iprange-to-cidr ips) |
  ipassmt-iprange-translate (Neg ips) = all-but-those-ips (concat (map ipt-iprange-to-cidr
ips))

definition to-ipassmt
  :: (iface × 'i::len ipt-iprange list negation-type) list ⇒ (iface × ('i word × nat)
list) list where
  to-ipassmt assmt = map (λ(ifce, ips). (ifce, ipassmt-iprange-translate ips)) assmt

definition zero-word ≡ 0 :: ('a :: len) word
```

export-code *Rule*
Match MatchNot MatchAnd MatchAny
Src Dst IIface OIface Prot Src-Ports Dst-Ports CT-State Extra
mk-L4Ports-pre
ProtoAny Proto TCP UDP ICMP L4-Protocol.IPv6ICMP L4-Protocol.SCTP L4-Protocol.GRE
L4-Protocol.ESP L4-Protocol.AH
Iface
integer-to-16word nat-to-16word nat-of-integer integer-of-nat word-less-eq word-to-nat

nat-to-8word
IpAddrNetmask IpAddrRange IpAddr
CT-New CT-Established CT-Related CT-Untracked CT-Invalid
TCP-Flags TCP-SYN TCP-ACK TCP-FIN TCP-RST TCP-URG TCP-PSH
Accept Drop Log Reject Call Return Goto Empty Unknown
action-toString

ipv4addr-of-dotdecimal
ipt-ipv4range-toString
common-primitive-ipv4-toString
common-primitive-match-expr-ipv4-toString
simple-rule-ipv4-toString

mk-ipv6addr IPv6AddrPreferred ipv6preferred-to-int int-to-ipv6preferred
ipt-ipv6range-toString
common-primitive-ipv6-toString
common-primitive-match-expr-ipv6-toString
simple-rule-ipv6-toString

Semantics-Goto.rewrite-Goto-safe
alist-and' compress-parsed-extra fill-l4-protocol Pos Neg mk-Set
unfold-ruleset-CHAIN-safe map-of-string
upper-closure
abstract-for-simple-firewall optimize-matches
packet-assume-new
to-simple-firewall
to-simple-firewall-without-interfaces
sanity-wf-ruleset
has-default-policy
ipassmt-generic-ipv4 ipassmt-generic-ipv6
no-spoofing-iface ipassmt-sanity-defined map-of-ipassmt to-ipassmt ipassmt-diff
Pos Neg

simple-fw-valid
debug-ipassmt-ipv4 debug-ipassmt-ipv6

access-matrix-pretty-ipv4 access-matrix-pretty-ipv6
mk-parts-connection-TCP

PrefixMatch routing-rule-ext routing-action-ext
routing-action-oiface-update metric-update routing-action-next-hop-update empty-rr-hlp sort-rtbl
prefix-match-32-toString routing-rule-32-toString prefix-match-128-toString routing-rule-128-toString
default-prefix sanity-ip-route ipassmt-diff routing-ipassmt
checking *SML Haskell?*

end
theory *Access-Matrix-Embeddings*
imports *Semantics-Embeddings*
Primitive-Matchers/No-Spoof
Simple-Firewall.Service-Matrix
begin

50 Applying the Access Matrix to the Bigstep Semantics

If the real iptables firewall (*iptables-bigstep*) accepts a packet, we have a corresponding edge in the *access-matrix*.

corollary *access-matrix-and-bigstep-semantics*:
defines *preprocess rs* ≡ *upper-closure* (*optimize-matches abstract-for-simple-firewall* (*upper-closure* (*packet-assume-new rs*)))
and *newpkt p* ≡ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*
fixes γ :: *'i*::*len common-primitive* ⇒ (*'i*, *'pkt-ext*) *tagged-packet-scheme* ⇒ *bool*
and *p* :: (*'i*::*len*, *'pkt-ext*) *tagged-packet-scheme*
assumes *agree*:*matcher-agree-on-exact-matches* γ *common-matcher*
and *simple*: *simple-ruleset rs*
and *new*: *newpkt p*
and *matrix*: (*V*,*E*) = *access-matrix* (⦇*pc-iiface* = *p-iiface p*, *pc-oiface* = *p-oiface p*, *pc-proto* = *p-proto p*, *pc-sport* = *p-sport p*, *pc-dport* = *p-dport p*⦈) (*to-simple-firewall* (*preprocess rs*))
and *accept*: Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*
shows ∃ *s-repr d-repr s-range d-range*. (*s-repr*, *d-repr*) ∈ *set E* ∧
(*map-of V*) *s-repr* = *Some s-range* ∧ (*p-src p*) ∈ *wordinterval-to-set s-range* ∧
(*map-of V*) *d-repr* = *Some d-range* ∧ (*p-dst p*) ∈ *wordinterval-to-set d-range*
proof −
let *?c*=⦇ *pc-iiface* = *p-iiface p*, *c-oiface* = *p-oiface p*, *pc-proto* = *p-proto p*, *pc-sport* = *p-sport p*, *pc-dport* = *p-dport p* ⦈
from *new-packets-to-simple-firewall-overapproximation*[*OF agree simple*] **have**
{*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow* ∧ *newpkt p*}
⊆
{*p*. *simple-fw* (*to-simple-firewall* (*preprocess rs*)) *p* = *Decision FinalAllow* ∧

newpkt p}
unfolding *preprocess-def newpkt-def* **by** *blast*
with *accept new* **have** *simple-fw* (*to-simple-firewall* (*preprocess rs*)) *p* = *Decision FinalAllow* **by** *blast*
hence *runFw-scheme* (*p-src p*) (*p-dst p*) *?c p* (*to-simple-firewall* (*preprocess rs*)) = *Decision FinalAllow*
by(*simp add*: *runFw-scheme-def*)
hence *runFw* (*p-src p*) (*p-dst p*) *?c* (*to-simple-firewall* (*preprocess rs*)) = *Decision FinalAllow*
by(*simp add*: *runFw-scheme*[*symmetric*])
with *access-matrix*[*OF matrix*] **show** *?thesis* **by** *presburger*
qed

corollary *access-matrix-no-interfaces-and-bigstep-semantics*:
defines *newpkt p* ≡ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*
fixes γ :: *'i*::*len common-primitive* ⇒ (*'i*, *'pkt-ext*) *tagged-packet-scheme* ⇒ *bool*
and *p* :: (*'i*::*len*, *'pkt-ext*) *tagged-packet-scheme*
assumes *agree*:*matcher-agree-on-exact-matches* γ *common-matcher*
and *simple*: *simple-ruleset rs*
— To get the best results, we want to rewrite all interfaces, which needs some preconditions

— well-formed ipassmt
and *wf-ipassmt1*: *ipassmt-sanity-nowildcards* (*map-of ipassmt*) **and** *wf-ipassmt2*: *distinct* (*map fst ipassmt*)
— There are no spoofed packets (probably by kernel's reverse path filter or our checker). This assumption implies that ipassmt lists ALL interfaces (!!).
and *nospoofing*: ∀ (*p*::(*'i*::*len*, *'pkt-ext*) *tagged-packet-scheme*).
∃ *ips*. (*map-of ipassmt*) (*Iface* (*p-iiface p*)) = *Some ips* ∧ *p-src p* ∈ *ipcidr-union-set* (*set ips*)
— If a routing table was passed, the output interface for any packet we consider is decided based on it.
and *routing-decided*: ⋀*rtbl* (*p*::(*'i*,*'pkt-ext*) *tagged-packet-scheme*). *rtblo* = *Some rtbl* ⟹ *output-iface* (*routing-table-semantics rtbl* (*p-dst p*)) = *p-oiface p*
— A passed routing table is wellformed
and *correct-routing*: ⋀*rtbl*. *rtblo* = *Some rtbl* ⟹ *correct-routing rtbl*
— A passed routing table contains no interfaces with wildcard names
and *routing-no-wildcards*: ⋀*rtbl*. *rtblo* = *Some rtbl* ⟹ *ipassmt-sanity-nowildcards* (*map-of* (*routing-ipassmt rtbl*))
and *new*: *newpkt p*
— building the matrix over ANY interfaces, not mentioned anywhere. That means, we don't care about interfaces!
and *matrix*: (*V*,*E*) = *access-matrix* (⦇*pc-iiface* = *anyI*, *pc-oiface* = *anyO*, *pc-proto* = *p-proto p*, *pc-sport* = *p-sport p*, *pc-dport* = *p-dport p*⦈
(*to-simple-firewall-without-interfaces ipassmt rtblo rs*)
and *accept*: Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*
shows ∃ *s-repr d-repr s-range d-range*. (*s-repr*, *d-repr*) ∈ *set E* ∧

```
            (map-of V) s-repr = Some s-range ∧ (p-src p) ∈ wordinterval-to-set
s-range ∧
            (map-of V) d-repr = Some d-range ∧ (p-dst p) ∈ wordinterval-to-set
d-range
proof −
  let ?c=⦇ pc-iiface = p-iiface p, c-oiface = p-oiface p, pc-proto = p-proto p,
         pc-sport = p-sport p, pc-dport = p-dport p ⦈
  let ?srs=to-simple-firewall-without-interfaces ipassmt rtblo rs
 note tosfw=to-simple-firewall-without-interfaces[OF simple wf-ipassmt1 wf-ipassmt2
nospoofing routing-decided correct-routing routing-no-wildcards, of rtblo, simplified]
  from tosfw(2) have no-ifaces: simple-firewall-without-interfaces ?srs unfolding
simple-firewall-without-interfaces-def by fastforce
  from simple simple-imp-good-ruleset have good-ruleset rs by blast
 with accept FinalAllow-approximating-in-doubt-allow[OF agree] have accept-ternary:
   (common-matcher, in-doubt-allow),p⊢ ⟨rs, Undecided⟩ ⇒α Decision FinalAllow
by blast
  from tosfw(1) have
   {p.(common-matcher, in-doubt-allow),p⊢ ⟨rs, Undecided⟩ ⇒α Decision FinalAl-
low ∧ newpkt p}
     ⊆
    {p. simple-fw ?srs p = Decision FinalAllow ∧ newpkt p}
    unfolding newpkt-def by blast
  with accept-ternary new have simple-fw ?srs p = Decision FinalAllow by blast
  hence runFw-scheme (p-src p) (p-dst p) ?c p ?srs = Decision FinalAllow
    by(simp add: runFw-scheme-def)
  hence runFw (p-src p) (p-dst p) ?c ?srs = Decision FinalAllow
    by(simp add: runFw-scheme[symmetric])
  hence runFw (p-src p) (p-dst p)
          ⦇pc-iiface = anyI, pc-oiface = anyO, pc-proto = p-proto p, pc-sport =
p-sport p, pc-dport = p-dport p⦈ ?srs = Decision FinalAllow
    apply(subst runFw-no-interfaces[OF no-ifaces]) by simp
  with access-matrix[OF matrix] show ?thesis by presburger
qed

end
theory Documentation
imports Semantics-Embeddings
    Call-Return-Unfolding
    No-Spoof-Embeddings
    Access-Matrix-Embeddings
    Primitive-Matchers/Code-Interface
begin
```

51 Documentation

51.1 General Model

The semantics of the filtering behavior of iptables is expressed by *iptables-bigstep*. The notation $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$ reads as follows: Γ is the

background ruleset (user-defined rules). γ is a function ($'a$, $'p$) *matcher* which is called the primitive matcher (i.e. the matching features supported by iptables). p is the packet inspected by the firewall. rs is the ruleset. s and t are the start state and final state.
The semantics:

$$\frac{}{\Gamma,\gamma,p\vdash_g \langle[],\ t\rangle \Rightarrow t}$$

$$\frac{Semantics\text{-}Goto.matches\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ action.Accept],\ Undecided\rangle \Rightarrow Decision\ FinalAllow}$$

$$\frac{Semantics\text{-}Goto.matches\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ action.Drop],\ Undecided\rangle \Rightarrow Decision\ FinalDeny}$$

$$\frac{Semantics\text{-}Goto.matches\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ Reject],\ Undecided\rangle \Rightarrow Decision\ FinalDeny}$$

$$\frac{Semantics\text{-}Goto.matches\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ Log],\ Undecided\rangle \Rightarrow Undecided}$$

$$\frac{Semantics\text{-}Goto.matches\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ Empty],\ Undecided\rangle \Rightarrow Undecided}$$

$$\frac{\neg\ Semantics\text{-}Goto.matches\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ a],\ Undecided\rangle \Rightarrow Undecided}$$

$$\Gamma,\gamma,p\vdash_g \langle rs,\ Decision\ X\rangle \Rightarrow Decision\ X$$

$$\frac{\Gamma,\gamma,p\vdash_g \langle rs_1,\ Undecided\rangle \Rightarrow t \qquad \Gamma,\gamma,p\vdash_g \langle rs_2,\ t\rangle \Rightarrow t' \qquad Semantics\text{-}Goto.no\text{-}matching\text{-}Goto\ \gamma\ p\ rs_1}{\Gamma,\gamma,p\vdash_g \langle rs_1\ @\ rs_2,\ Undecided\rangle \Rightarrow t'}$$

$$\frac{\begin{array}{c}Semantics\text{-}Goto.matches\ \gamma\ m\ p\\ \Gamma\ chain = Some\ (rs_1\ @\ [Rule\ m'\ Return]\ @\ rs_2)\\ Semantics\text{-}Goto.matches\ \gamma\ m'\ p\\ \Gamma,\gamma,p\vdash_g \langle rs_1,\ Undecided\rangle \Rightarrow Undecided\\ Semantics\text{-}Goto.no\text{-}matching\text{-}Goto\ \gamma\ p\ rs_1\end{array}}{\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow Undecided}$$

$$\frac{Semantics\text{-}Goto.matches\ \gamma\ m\ p \qquad \Gamma\ chain = Some\ rs \qquad \Gamma,\gamma,p\vdash_g \langle rs,\ Undecided\rangle \Rightarrow t}{\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow t}$$

51.2 Unfolding the Ruleset

We can replace all *Goto*s to terminal chains (chains that ultimately yield a final decision for every packet) with *Call*s. Otherwise we don't have as rich

goto semantics as iptables has, but this rewriting is safe.

Semantics-Goto.rewrite-Goto-chain-safe Γ *rs* = *Some rs'* $\Longrightarrow$ Γ,γ,*p*$\vdash_g$ $\langle$*rs'*, *s*$\rangle$ $\Rightarrow$ *t* = Γ,γ,*p*$\vdash_g$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow$ *t*

The iptables firewall starts as follows: [*Rule MatchAny* (*Call chain-name*), *Rule MatchAny default-action*] We call to a built-in chain *chain-name*, usually INPUT, OUTPUT, or FORWARD. If we don't get a decision, iptables uses the default policy (-P) *default-action*.

We can call *unfold-optimize-ruleset-CHAIN* to remove all calls to user-defined chains and other unpleasant actions. We get back a *simple-ruleset* which as exactly the same behaviour. As a bonus, this *simple-ruleset* already has some match conditions optimized.

For example, if the parser does not find a source IP in a rule, it is okay to specify -s 0.0.0.0/0, the unfolding will optimize away these things for you. Or if you parse iptables -L -n which always has these annoying 0.0.0.0/0 fields. May make the parser easier. The following lemma shows that this does not change the semantics.

lemma *unfold-optimize-common-matcher-univ-ruleset-CHAIN*:
— for IPv4 and IPv6 packets
fixes γ :: *'i*::*len common-primitive* $\Rightarrow$ (*'i*, *'pkt-ext*) *tagged-packet-scheme* $\Rightarrow$ *bool*
and *p* :: (*'i*::*len*, *'pkt-ext*) *tagged-packet-scheme*
assumes *sanity-wf-ruleset* Γ **and** *chain-name* $\in$ *set* (*map fst* Γ)
and *default-action* = *action.Accept* $\vee$ *default-action* = *action.Drop*
and *matcher-agree-on-exact-matches* γ *common-matcher*
and *unfold-ruleset-CHAIN-safe chain-name default-action* (*map-of* Γ) = *Some rs*
shows (*map-of* Γ),γ,*p*$\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow$ *t* $\longleftrightarrow$
(*map-of* Γ),γ,*p*$\vdash$ $\langle$[*Rule MatchAny* (*Call chain-name*), *Rule MatchAny default-action*], *s*$\rangle$ $\Rightarrow$ *t*
and *simple-ruleset rs*
apply(*intro unfold-optimize-ruleset-CHAIN*[**where** *optimize*=*optimize-primitive-univ*, *OF assms*(*1*) *assms*(*2*) *assms*(*3*)])
using *assms* **apply**(*simp-all add*: *unfold-ruleset-CHAIN-safe-def Semantics-optimize-primitive-univ-common-*
by(*simp add*: *unfold-optimize-ruleset-CHAIN-def Let-def split*: *if-split-asm*)

51.3 Spoofing protection

We provide an executable algorithm *no-spoofing-iface* which checks that a ruleset provides spoofing protection:

⟦*matcher-agree-on-exact-matches* γ *common-matcher*; *simple-ruleset rs*; $\forall$ *r*$\in$*set rs*. *normalized-nnf-match* (*get-match r*); $\forall$ *iface*$\in$*dom ipassmt*. *no-spoofing-iface iface ipassmt rs*; *iface* $\in$ *dom ipassmt*⟧ $\Longrightarrow$ {*p-src p* | Γ,γ,*p*⦇*p-iiface* := *iface-sel iface*⦈$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow*} $\subseteq$ *ipcidr-union-set* (*set* (*the* (*ipassmt iface*)))

Text the firewall needs normalized match conditions, this is a good way to

preprocess the firewall before checking spoofing protection:

⟦*matcher-agree-on-exact-matches* γ *common-matcher*; *simple-ruleset rs*; ∀ *iface*∈*dom ipassmt*. *no-spoofing-iface iface ipassmt* (*upper-closure* (*packet-assume-new rs*)); *iface* ∈ *dom ipassmt*⟧ ⟹ {*p-src p* | (*match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*) ∧ Γ,γ,*p*⟦*p-iiface* := *iface-sel iface*⟧⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*} ⊆ *ipcidr-union-set* (*set* (*the* (*ipassmt iface*)))

51.4 Simple Firewall Model

The simple firewall supports the following match conditions: *'i simple-match*.

The *simple-fw* model is remarkably simple: *simple-fw* [] *uu* = *Undecided*

simple-fw (*SimpleRule m simple-action.Accept* · *rs*) *p* = (*if simple-matches m p then Decision FinalAllow else simple-fw rs p*)

simple-fw (*SimpleRule m simple-action.Drop* · *rs*) *p* = (*if simple-matches m p then Decision FinalDeny else simple-fw rs p*)

We support translating to a stricter version (a version that accepts less packets):

⟦*matcher-agree-on-exact-matches* γ *common-matcher*; *simple-ruleset rs*⟧ ⟹ {*p* | *simple-fw* (*to-simple-firewall* (*lower-closure* (*optimize-matches abstract-for-simple-firewall* (*lower-closure* (*packet-assume-new rs*))))) *p* = *Decision FinalAllow* ∧ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*} ⊆ {*p* | Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow* ∧ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*}

We support translating to a more permissive version (a version that accepts more packets):

⟦*matcher-agree-on-exact-matches* γ *common-matcher*; *simple-ruleset rs*⟧ ⟹ {*p* | Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow* ∧ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*} ⊆ {*p* | *simple-fw* (*to-simple-firewall* (*upper-closure* (*optimize-matches abstract-for-simple-firewall* (*upper-closure* (*packet-assume-new rs*))))) *p* = *Decision FinalAllow* ∧ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*}

There is also a different approach to translate to the simple firewall which removes all matches on interfaces:

⟦*simple-ruleset rs*; *ipassmt-sanity-nowildcards* (*map-of ipassmt*); *distinct* (*map fst ipassmt*); ∀ *p*. ∃ *ips*. *map-of ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* ∧ *p-src p* ∈ *ipcidr-union-set* (*set ips*); ⋀*rtbl p*. *rtblo* = *Some rtbl* ⟹ *output-iface* (*routing-table-semantics rtbl* (*p-dst p*)) = *p-oiface p*; ⋀*rtbl*. *rtblo* = *Some rtbl* ⟹ *correct-routing rtbl*; ⋀*rtbl*. *rtblo* = *Some rtbl* ⟹ *ipassmt-sanity-nowildcards* (*map-of* (*routing-ipassmt rtbl*))⟧ ⟹ {*p* | (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒$_\alpha$ *Decision FinalAllow* ∧ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*} ⊆ {*p* | *simple-fw*

(*to-simple-firewall-without-interfaces ipassmt rtblo rs*) *p* = *Decision FinalAllow* ∧ *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*}
⟦*simple-ruleset rs*; *ipassmt-sanity-nowildcards* (*map-of ipassmt*); *distinct* (*map fst ipassmt*); ∀ *p*. ∃ *ips*. *map-of ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* ∧ *p-src p* ∈ *ipcidr-union-set* (*set ips*); ⋀*rtbl p*. *rtblo* = *Some rtbl* ⟹ *output-iface* (*routing-table-semantics rtbl* (*p-dst p*)) = *p-oiface p*; ⋀*rtbl*. *rtblo* = *Some rtbl* ⟹ *correct-routing rtbl*; ⋀*rtbl*. *rtblo* = *Some rtbl* ⟹ *ipassmt-sanity-nowildcards* (*map-of* (*routing-ipassmt rtbl*))⟧ ⟹ ∀ *r*∈*set* (*to-simple-firewall-without-in*
ipassmt rtblo rs). *iiface* (*match-sel r*) = *ifaceAny* ∧ *oiface* (*match-sel r*) = *ifaceAny*

51.5 Service Matrices

For a *′i simple-rule list* and a fixed *parts-connection*, we support to partition the IPv4 address space the following.

All members of a partition have the same access rights: *V* ∈ *set* (*build-ip-partition c rs*) ⟹ ∀ *ip1*∈*wordinterval-to-set V*. ∀ *ip2*∈*wordinterval-to-set V*. *same-fw-behaviour-one ip1 ip2 c rs*

Minimal: ⟦*A* ∈ *set* (*build-ip-partition c rs*); *B* ∈ *set* (*build-ip-partition c rs*); *A* ≠ *B*⟧ ⟹ ∀ *ip1*∈*wordinterval-to-set A*. ∀ *ip2*∈*wordinterval-to-set B*. ¬ *same-fw-behaviour-one ip1 ip2 c rs*

The resulting access control matrix is sound and complete:
(*V*, *E*) = *access-matrix c rs* ⟹ (∃ *s-repr d-repr s-range d-range*. (*s-repr*, *d-repr*) ∈ *set E* ∧ *map-of V s-repr* = *Some s-range* ∧ *s* ∈ *wordinterval-to-set s-range* ∧ *map-of V d-repr* = *Some d-range* ∧ *d* ∈ *wordinterval-to-set d-range*) = (*runFw s d c rs* = *Decision FinalAllow*)

Theorem reads: For a fixed connection, you can look up IP addresses (source and destination pairs) in the matrix if and only if the firewall accepts this src,dst IP address pair for the fixed connection. Note: The matrix is actually a graph (nice visualization!), you need to look up IP addresses in the Vertices and check the access of the representants in the edges. If you want to visualize the graph (e.g. with Graphviz or tkiz): The vertices are the node description (i.e. header; *dom V* is the label for each node which will also be referenced in the edges, *ran V* is the human-readable description for each node (i.e. the full IP range it represents)), the edges are the edges. Result looks nice. Theorem also tells us that this visualization is correct.

A final theorem which does not mention the simple firewall at all. If the real iptables firewall (*iptables-bigstep*) accepts a packet, we have a corresponding edge in the *access-matrix*:

⟦*matcher-agree-on-exact-matches* γ *common-matcher*; *simple-ruleset rs*; *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ∧ *p-tag-ctstate p* = *CT-New*; (*V*, *E*) = *access-matrix* ⦇*pc-iiface* = *p-iiface p*, *pc-oiface* = *p-oiface p*, *pc-proto* = *p-proto p*, *pc-sport*

$= p\text{-}sport\ p,\ pc\text{-}dport = p\text{-}dport\ p\rparr\ (to\text{-}simple\text{-}firewall\ (upper\text{-}closure\ (optimize\text{-}matches\ abstract\text{-}for\text{-}simple\text{-}firewall\ (upper\text{-}closure\ (packet\text{-}assume\text{-}new\ rs)))))$; $\Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Decision\ FinalAllow\rrbracket \Longrightarrow \exists\ s\text{-}repr\ d\text{-}repr\ s\text{-}range\ d\text{-}range.\ (s\text{-}repr,\ d\text{-}repr) \in set\ E \wedge map\text{-}of\ V\ s\text{-}repr = Some\ s\text{-}range \wedge p\text{-}src\ p \in wordinterval\text{-}to\text{-}set\ s\text{-}range \wedge map\text{-}of\ V\ d\text{-}repr = Some\ d\text{-}range \wedge p\text{-}dst\ p \in wordinterval\text{-}to\text{-}set\ d\text{-}range$

Actually, we want to ignore all interfaces for a service matrix. This is done in $\llbracket matcher\text{-}agree\text{-}on\text{-}exact\text{-}matches\ \gamma\ common\text{-}matcher;\ simple\text{-}ruleset\ rs;\ ipassmt\text{-}sanity\text{-}nowildcards\ (map\text{-}of\ ipassmt);\ distinct\ (map\ fst\ ipassmt);\ \forall\ p.\ \exists\ ips.\ map\text{-}of\ ipassmt\ (Iface\ (p\text{-}iiface\ p)) = Some\ ips \wedge p\text{-}src\ p \in ipcidr\text{-}union\text{-}set\ (set\ ips);\ \bigwedge rtbl\ p.\ rtblo = Some\ rtbl \Longrightarrow output\text{-}iface\ (routing\text{-}table\text{-}semantics\ rtbl\ (p\text{-}dst\ p)) = p\text{-}oiface\ p;\ \bigwedge rtbl.\ rtblo = Some\ rtbl \Longrightarrow correct\text{-}routing\ rtbl;\ \bigwedge rtbl.\ rtblo = Some\ rtbl \Longrightarrow ipassmt\text{-}sanity\text{-}nowildcards\ (map\text{-}of\ (routing\text{-}ipassmt\ rtbl));\ match\text{-}tcp\text{-}flags\ ipt\text{-}tcp\text{-}syn\ (p\text{-}tcp\text{-}flags\ p) \wedge p\text{-}tag\text{-}ctstate\ p = CT\text{-}New;\ (V,\ E) = access\text{-}matrix\ \llparr pc\text{-}iiface = anyI,\ pc\text{-}oiface = anyO,\ pc\text{-}proto = p\text{-}proto\ p,\ pc\text{-}sport = p\text{-}sport\ p,\ pc\text{-}dport = p\text{-}dport\ p\rparr\ (to\text{-}simple\text{-}firewall\text{-}without\text{-}interfaces\ ipassmt\ rtblo\ rs);\ \Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Decision\ FinalAllow\rrbracket \Longrightarrow \exists\ s\text{-}repr\ d\text{-}repr\ s\text{-}range\ d\text{-}range.\ (s\text{-}repr,\ d\text{-}repr) \in set\ E \wedge map\text{-}of\ V\ s\text{-}repr = Some\ s\text{-}range \wedge p\text{-}src\ p \in wordinterval\text{-}to\text{-}set\ s\text{-}range \wedge map\text{-}of\ V\ d\text{-}repr = Some\ d\text{-}range \wedge p\text{-}dst\ p \in wordinterval\text{-}to\text{-}set\ d\text{-}range$. The theorem reads a bit ugly because we need well-formedness assumptions if we rewrite interfaces. Internally, it uses *iface-try-rewrite* which is pretty safe to use, even if you don't have an *ipassmt* or routing tables.

end

References

[1] C. Diekmann. Verified Firewall Ruleset Verification – Math, Functional Programming, Theorem Proving, and an Introduction to Isabelle/HOL. 32. Chaos Communication Congress (32C3), 12 2015. Recording: https://media.ccc.de/v/32c3-7195-verified_firewall_ruleset_verification.

[2] C. Diekmann, L. Hupel, and G. Carle. Semantics-Preserving Simplification of Real-World Firewall Rule Sets. In *20th International Symposium on Formal Methods*, pages 195–212. Springer, 6 2015. http://www.net.in.tum.de/fileadmin/bibtex/publications/papers/fm15_Semantics-Preserving_Simplification_of_Real-World_Firewall_Rule_Sets.pdf.

[3] C. Diekmann, J. Michaelis, M. Haslbeck, and G. Carle. Verified iptables Firewall Analysis. In *IFIP Networking 2016*, Vienna, Austria, 5 2016. http://dl.ifip.org/db/conf/networking/networking2016/1570232858.pdf.

[4] C. Diekmann, L. Schwaighofer, and G. Carle. Certifying Spoofing-Protection of Firewalls. In *11th International Conference on Network and Service Management, CNSM*, Barcelona, Spain, 11 2015. http://www.net.in.tum.de/fileadmin/bibtex/publications/papers/diekmann2015_cnsm.pdf.