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

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*⟩

lemma *repeat-stabilize-induct*: $(P\ m) \Longrightarrow (\bigwedge m.\ P\ m \Longrightarrow P\ (f\ m)) \Longrightarrow P\ (\mathit{repeat\text{-}stabilize}\ n\ f\ m)$
⟨*proof*⟩

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*:
$[\mathit{Rule}\ m\ a] = rs_1\ @\ rs_2 \Longrightarrow$
$(rs_1 = [\mathit{Rule}\ m\ a] \Longrightarrow rs_2 = [] \Longrightarrow P\ m\ a) \Longrightarrow$
$(rs_1 = [] \Longrightarrow rs_2 = [\mathit{Rule}\ m\ a] \Longrightarrow P\ m\ a) \Longrightarrow P\ m\ a$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rm-LogEmpty-seq*: *rm-LogEmpty* (*rs1*@*rs2*) = *rm-LogEmpty rs1* @ *rm-LogEmpty rs2*
⟨*proof*⟩

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*) ⟨*proof*⟩
lemma *m* = (*MatchAnd* (*MatchAnd MatchAny MatchAny*) (*MatchAnd MatchAny MatchAny*)) ⟹
(*opt-MatchAny-match-expr-once*^^*2*) *m* ≠ *opt-MatchAny-match-expr-once m* ⟨*proof*⟩

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* ⟨*proof*⟩

lemma *simple-ruleset-append*: *simple-ruleset* (rs_1 @ rs_2) ⟷ *simple-ruleset* rs_1 ∧ *simple-ruleset* rs_2
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *matcheq-matchNone-no-primitive*: ¬ *has-primitive m* ⟹ *matcheq-matchNone* (*MatchNot m*) ⟷ ¬ *matcheq-matchNone m*

⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *optimize-matches-option-append*: *optimize-matches-option f* (*rs1*@*rs2*) = *optimize-matches-option f rs1* @ *optimize-matches-option f rs2*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *optimize-matches-fst*: *optimize-matches f* (*r*#*rs*) = *optimize-matches f* [*r*]@*optimize-matches f rs*
⟨*proof*⟩

lemma *optimize-matches-preserves*: (⋀ *r*. *r* ∈ *set rs* ⟹ *P* (*f* (*get-match r*))) ⟹
∀ *r* ∈ *set* (*optimize-matches f rs*). *P* (*get-match r*)
⟨*proof*⟩

lemma *optimize-matches-simple-ruleset*: *simple-ruleset rs* ⟹ *simple-ruleset* (*optimize-matches f rs*)
⟨*proof*⟩

definition *optimize-matches-a* :: (*action* ⇒ *'a match-expr* ⇒ *'a match-expr*) ⇒ *'a rule list* ⇒ *'a rule list* **where**

optimize-matches-a f rs = *map* (*λr*. *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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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⊢ ⟨rs_1, *Undecided*⟩ ⇒ t; Γ,γ,p⊢ ⟨rs_2, t⟩ ⇒ t'⟧ ⟹ Γ,γ,p⊢ ⟨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⊢ ⟨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:

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

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

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

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

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

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

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

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

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

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

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

lemma *deny*:
$matches\ \gamma\ m\ p \Longrightarrow a = Drop \lor a = Reject \Longrightarrow iptables\text{-}bigstep\ \Gamma\ \gamma\ p\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalDeny)$
⟨*proof*⟩

lemma *seq-cons*:
assumes $\Gamma,\gamma,p\vdash\ \langle[r],Undecided\rangle\ \Rightarrow\ t$ **and** $\Gamma,\gamma,p\vdash\ \langle rs,t\rangle\ \Rightarrow\ t'$
shows $\Gamma,\gamma,p\vdash\ \langle r\#rs,\ Undecided\rangle\ \Rightarrow\ t'$
⟨*proof*⟩

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

$\Longrightarrow \Gamma,\gamma,p\vdash \langle rs, Undecided\rangle \Rightarrow t \Longrightarrow P\ rs\ Undecided\ t \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ t\ \rrbracket \Longrightarrow$
$P\ rs\ s\ t$
⟨*proof*⟩

lemma *skipD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [] \Longrightarrow s = t$
⟨*proof*⟩

lemma *decisionD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow s = Decision\ X \Longrightarrow t = Decision\ X$
⟨*proof*⟩

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

lemma *acceptD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Accept] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Decision\ FinalAllow$
⟨*proof*⟩

lemma *dropD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Drop] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Decision\ FinalDeny$
⟨*proof*⟩

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

lemma *logD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Log] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Undecided$
⟨*proof*⟩

lemma *emptyD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Empty] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Undecided$
⟨*proof*⟩

lemma *nomatchD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ a] \Longrightarrow s = Undecided \Longrightarrow \neg\ matches\ \gamma\ m\ p \Longrightarrow t = Undecided$
⟨*proof*⟩

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** $rs = rs_1\ @\ Rule\ m'\ Return\ \#\ rs_2$ $matches\ \gamma\ m'\ p$ $\Gamma,\gamma,p\vdash \langle rs_1,s\rangle \Rightarrow Undecided$ $t = Undecided$
⟨*proof*⟩

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 $\Gamma,\gamma,p\vdash \langle rs,s\rangle \Rightarrow t'$
⟨*proof*⟩

lemma *seq'-cons*: $\Gamma,\gamma,p\vdash \langle [r],s\rangle \Rightarrow t \Longrightarrow \Gamma,\gamma,p\vdash \langle rs,t\rangle \Rightarrow t' \Longrightarrow \Gamma,\gamma,p\vdash \langle r\#rs,\ s\rangle \Rightarrow t'$
⟨*proof*⟩

lemma *seq-split*:
assumes $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$ $rs = rs_1@rs_2$
obtains t' **where** $\Gamma,\gamma,p\vdash \langle rs_1,s\rangle \Rightarrow t'$ $\Gamma,\gamma,p\vdash \langle rs_2,t'\rangle \Rightarrow t$
⟨*proof*⟩

lemma *seqE*:
assumes $\Gamma,\gamma,p\vdash \langle rs_1@rs_2,\ s\rangle \Rightarrow t$
obtains *ti* **where** $\Gamma,\gamma,p\vdash \langle rs_1,s\rangle \Rightarrow ti$ $\Gamma,\gamma,p\vdash \langle rs_2,ti\rangle \Rightarrow t$
⟨*proof*⟩

lemma *seqE-cons*:
assumes $\Gamma,\gamma,p\vdash \langle r\#rs,\ s\rangle \Rightarrow t$
obtains *ti* **where** $\Gamma,\gamma,p\vdash \langle [r],s\rangle \Rightarrow ti$ $\Gamma,\gamma,p\vdash \langle rs,ti\rangle \Rightarrow t$
⟨*proof*⟩

lemma *nomatch'*:
assumes $\bigwedge r.\ r \in set\ rs \Longrightarrow \neg\ matches\ \gamma\ (get\text{-}match\ r)\ p$
shows $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow s$
⟨*proof*⟩

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*⟩

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*⟩

theorem *iptables-bigstep-deterministic*: **assumes** $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$ **and** $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t'$ **shows** $t = t'$
⟨*proof*⟩

lemma *iptables-bigstep-to-undecided*: $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow \mathit{Undecided} \Longrightarrow s = \mathit{Undecided}$
⟨*proof*⟩

lemma *iptables-bigstep-to-decision*: $\Gamma,\gamma,p\vdash \langle rs,\ \mathit{Decision}\ Y\rangle \Rightarrow \mathit{Decision}\ X \Longrightarrow Y = X$
⟨*proof*⟩

lemma *Rule-UndecidedE*:
assumes $\Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ a],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}$
obtains (*nomatch*) $\neg$ *matches* γ *m p*
| (*log*) $a = \mathit{Log} \vee a = \mathit{Empty}$
| (*call*) *c* **where** $a = \mathit{Call}\ c$ *matches* γ *m p*
⟨*proof*⟩

lemma *Rule-DecisionE*:
assumes $\Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ a],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ X$
obtains (*call*) *chain* **where** *matches* γ *m p* $a = \mathit{Call\ chain}$
| (*accept-reject*) *matches* γ *m p* $X = \mathit{FinalAllow} \Longrightarrow a = \mathit{Accept}$ $X = \mathit{FinalDeny} \Longrightarrow a = \mathit{Drop} \vee a = \mathit{Reject}$
⟨*proof*⟩

lemma *log-remove*:
assumes $\Gamma,\gamma,p\vdash \langle rs_1\ @\ [\mathit{Rule}\ m\ \mathit{Log}]\ @\ rs_2,\ s\rangle \Rightarrow t$
shows $\Gamma,\gamma,p\vdash \langle rs_1\ @\ rs_2,\ s\rangle \Rightarrow t$
⟨*proof*⟩
lemma *empty-empty*:
assumes $\Gamma,\gamma,p\vdash \langle rs_1\ @\ [\mathit{Rule}\ m\ \mathit{Empty}]\ @\ rs_2,\ s\rangle \Rightarrow t$
shows $\Gamma,\gamma,p\vdash \langle rs_1\ @\ rs_2,\ s\rangle \Rightarrow t$
⟨*proof*⟩

lemma *Unknown-actions-False*: $\Gamma,\gamma,p\vdash \langle r\ \#\ rs,\ \mathit{Undecided}\rangle \Rightarrow t \Longrightarrow r = \mathit{Rule}\ m\ a \Longrightarrow \mathit{matches}\ \gamma\ m\ p \Longrightarrow a = \mathit{Unknown} \vee (\exists\, chain.\ a = \mathit{Goto}\ chain) \Longrightarrow \mathit{False}$
⟨*proof*⟩

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* γ *m p* ⟹ *p*⊢' ⟨[*Rule m Log*], *Undecided*⟩ ⇒ *Undecided* |
empty: *matches* γ *m p* ⟹ *p*⊢' ⟨[*Rule m Empty*], *Undecided*⟩ ⇒ *Undecided* |
nomatch: ¬ *matches* γ *m p* ⟹ *p*⊢' ⟨[*Rule m a*], *Undecided*⟩ ⇒ *Undecided* |
decision: *p*⊢' ⟨*rs*, *Decision X*⟩ ⇒ *Decision X* |
seq: ⟦*p*⊢' ⟨rs_1, *Undecided*⟩ ⇒ *t*; *p*⊢' ⟨rs_2, *t*⟩ ⇒ *t'*⟧ ⟹ *p*⊢' ⟨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*; *p*⊢' ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* ⟧ ⟹
p⊢' ⟨[*Rule m* (*Call chain*)], *Undecided*⟩ ⇒ *Undecided* |
call-result: ⟦ *matches* γ *m p*; *p*⊢' ⟨*the* (Γ *chain*), *Undecided*⟩ ⇒ *t* ⟧ ⟹
p⊢' ⟨[*Rule m* (*Call chain*)], *Undecided*⟩ ⇒ *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*
⟨*proof*⟩
lemma *wf*-Γ-*tail*: *wf*-Γ (*r* # *rs*) ⟹ *wf*-Γ *rs* ⟨*proof*⟩
lemma *wf*-Γ-*Call*: *wf*-Γ [*Rule m* (*Call chain*)] ⟹ *wf*-Γ (*the* (Γ *chain*)) ∧ (∃ *rs*. Γ *chain* = *Some rs*)
⟨*proof*⟩

lemma *wf*-Γ *rs* ⟹ *p*⊢' ⟨*rs*, *s*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t*
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *wf-chain-fst*: *wf-chain* Γ (*r* # *rs*) ⟹ *wf-chain* Γ (*rs*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*]
⟨*proof*⟩

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*⟩

lemma *semantics-bigstep-defined1*: **assumes** $\forall$ *rsg* $\in$ *ran* $\Gamma \cup \{$*rs*$\}$. *wf-chain* Γ *rsg*
and $\forall$ *rsg* $\in$ *ran* $\Gamma \cup \{$*rs*$\}$. $\forall$ *r* $\in$ *set rsg*. ($\forall$ *chain*. *get-action r* $\neq$ *Goto chain*) $\wedge$ *get-action r* $\neq$ *Unknown*
and $\forall$ *r* $\in$ *set rs*. *get-action r* $\neq$ *Return*
and ($\forall$ *name* $\in$ *dom* Γ. $\exists$ *t*. $\Gamma,\gamma,p\vdash \langle$*the* ($\Gamma$ *name*), *Undecided*$\rangle \Rightarrow$ *t*)
shows $\exists$ *t*. $\Gamma,\gamma,p\vdash \langle$*rs*, *s*$\rangle \Rightarrow$ *t*
$\langle$*proof*$\rangle$

Showing the main theorem

context
begin
private lemma *iptables-bigstep-defined-if-singleton-rules*:
$\forall$ *r* $\in$ *set rs*. ($\exists$ *t*. $\Gamma,\gamma,p\vdash \langle$[*r*], *s*$\rangle \Rightarrow$ *t*) $\Longrightarrow \exists$ *t*. $\Gamma,\gamma,p\vdash \langle$*rs*, *s*$\rangle \Rightarrow$ *t*
$\langle$*proof*$\rangle$

well founded relation.

definition *calls-chain* :: *'a ruleset* $\Rightarrow$ (*string* $\times$ *string*) *set* **where**
calls-chain Γ = {(*r*, *s*). *case* Γ *r of Some rs* $\Rightarrow \exists$ *m*. *Rule m* (*Call s*) $\in$ *set rs* | *None* $\Rightarrow$ *False*}

lemma *calls-chain-def2*: *calls-chain* Γ = {(*caller*, *callee*). $\exists$ *rs m*. Γ *caller* = *Some rs* $\wedge$ *Rule m* (*Call callee*) $\in$ *set rs*}
$\langle$*proof*$\rangle$

example

private lemma *calls-chain* [
''FORWARD'' $\mapsto$ [(*Rule m1 Log*), (*Rule m2* (*Call ''foo''*)), (*Rule m3 Accept*), (*Rule m'* (*Call ''baz''*))],
''foo'' $\mapsto$ [(*Rule m4 Log*), (*Rule m5 Return*), (*Rule m6* (*Call ''bar''*))],
''bar'' $\mapsto$ [],
''baz'' $\mapsto$ []] =
{(*''FORWARD''*, *''foo''*), (*''FORWARD''*, *''baz''*), (*''foo''*, *''bar''*)}
$\langle$*proof*$\rangle$ **lemma** *wf* (*calls-chain* [
''FORWARD'' $\mapsto$ [(*Rule m1 Log*), (*Rule m2* (*Call ''foo''*)), (*Rule m3 Accept*), (*Rule m'* (*Call ''baz''*))],
''foo'' $\mapsto$ [(*Rule m4 Log*), (*Rule m5 Return*), (*Rule m6* (*Call ''bar''*))],
''bar'' $\mapsto$ [],
''baz'' $\mapsto$ []])
$\langle$*proof*$\rangle$

In our proof, we will need the reverse.

private definition *called-by-chain* :: *'a ruleset* $\Rightarrow$ (*string* $\times$ *string*) *set* **where**
called-by-chain Γ = {(*callee*, *caller*). *case* Γ *caller of Some rs* $\Rightarrow \exists$ *m*. *Rule m* (*Call callee*) $\in$ *set rs* | *None* $\Rightarrow$ *False*}
private lemma *called-by-chain-converse*: *calls-chain* Γ = *converse* (*called-by-chain* Γ)
$\langle$*proof*$\rangle$ **lemma** *wf-called-by-chain*: *finite* (*calls-chain* Γ) $\Longrightarrow$ *wf* (*calls-chain* Γ) $\Longrightarrow$ *wf* (*called-by-chain* Γ)

$\langle proof \rangle$ **lemma** *helper-cases-call-subchain-defined-or-return*:
$(\forall x{\in}ran\ \Gamma.\ wf\text{-}chain\ \Gamma\ x) \Longrightarrow$
$\forall rsg{\in}ran\ \Gamma.\ \forall r{\in}set\ rsg.\ (\forall chain.\ get\text{-}action\ r \neq Goto\ chain) \wedge get\text{-}action\ r \neq Unknown \Longrightarrow$
$\forall y\ m.\ \forall r{\in}set\ rs\text{-}called.\ r = Rule\ m\ (Call\ y) \longrightarrow (\exists t.\ \Gamma,\gamma,p\vdash \langle[Rule\ m\ (Call\ y)],\ Undecided\rangle \Rightarrow t) \Longrightarrow$
$wf\text{-}chain\ \Gamma\ rs\text{-}called \Longrightarrow$
$\forall r{\in}set\ rs\text{-}called.\ (\forall chain.\ get\text{-}action\ r \neq Goto\ chain) \wedge get\text{-}action\ r \neq Unknown \Longrightarrow$
$(\exists t.\ \Gamma,\gamma,p\vdash \langle rs\text{-}called,\ Undecided\rangle \Rightarrow t) \vee$
$(\exists rs\text{-}called1\ rs\text{-}called2\ m'.$
$rs\text{-}called = (rs\text{-}called1\ @\ [Rule\ m'\ Return]\ @\ rs\text{-}called2) \wedge$
$matches\ \gamma\ m'\ p \wedge \Gamma,\gamma,p\vdash \langle rs\text{-}called1,\ Undecided\rangle \Rightarrow Undecided)$
$\langle proof \rangle$

lemma *helper-defined-single*:
assumes $wf\ (called\text{-}by\text{-}chain\ \Gamma)$
and $\forall rsg \in ran\ \Gamma \cup \{[Rule\ m\ a]\}.\ wf\text{-}chain\ \Gamma\ rsg$
and $\forall rsg \in ran\ \Gamma \cup \{[Rule\ m\ a]\}.\ \forall\ r \in set\ rsg.\ (\neg(\exists chain.\ get\text{-}action\ r = Goto\ chain)) \wedge get\text{-}action\ r \neq Unknown$
and $a \neq Return$
shows $\exists t.\ \Gamma,\gamma,p\vdash \langle[Rule\ m\ a],\ s\rangle \Rightarrow t$
$\langle proof \rangle$ **lemma** *helper-defined-ruleset-calledby*: $wf\ (called\text{-}by\text{-}chain\ \Gamma) \Longrightarrow$
$\forall rsg \in ran\ \Gamma \cup \{rs\}.\ wf\text{-}chain\ \Gamma\ rsg \Longrightarrow$
$\forall rsg \in ran\ \Gamma \cup \{rs\}.\ \forall\ r \in set\ rsg.\ (\neg(\exists chain.\ get\text{-}action\ r = Goto\ chain)) \wedge get\text{-}action\ r \neq Unknown \Longrightarrow$
$\forall\ r \in set\ rs.\ get\text{-}action\ r \neq Return \Longrightarrow$
$\exists t.\ \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
$\langle proof \rangle$

corollary *semantics-bigstep-defined*: $finite\ (calls\text{-}chain\ \Gamma) \Longrightarrow wf\ (calls\text{-}chain\ \Gamma) \Longrightarrow$ — call relation finite and terminating
$\forall rsg \in ran\ \Gamma \cup \{rs\}.\ wf\text{-}chain\ \Gamma\ rsg \Longrightarrow$ — All calls to defined chains
$\forall rsg \in ran\ \Gamma \cup \{rs\}.\ \forall\ r \in set\ rsg.\ (\forall x.\ get\text{-}action\ r \neq Goto\ x) \wedge get\text{-}action\ r \neq Unknown \Longrightarrow$ — no bad actions
$\forall\ r \in set\ rs.\ get\text{-}action\ r \neq Return$ — no toplevel return $\Longrightarrow$
$\exists t.\ \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
$\langle proof \rangle$
end

Common Algorithms

lemma *iptables-bigstep-rm-LogEmpty*: $\Gamma,\gamma,p\vdash \langle rm\text{-}LogEmpty\ rs,\ s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
$\langle proof \rangle$

lemma *iptables-bigstep-rw-Reject*: $\Gamma,\gamma,p\vdash \langle rw\text{-}Reject\ rs,\ s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
$\langle proof \rangle$

end
theory *Matching*
imports *Semantics*
begin

4.1 Boolean Matcher Algebra

lemma *MatchOr*: *matches* γ (*MatchOr m1 m2*) *p* $\longleftrightarrow$ *matches* γ *m1 p* $\vee$ *matches* γ *m2 p*
⟨*proof*⟩

lemma *opt-MatchAny-match-expr-correct*: *matches* γ (*opt-MatchAny-match-expr m*) = *matches* γ *m*
⟨*proof*⟩

lemma *matcheq-matchAny*: $\neg$ *has-primitive m* $\Longrightarrow$ *matcheq-matchAny m* $\longleftrightarrow$ *matches* γ *m p*
⟨*proof*⟩

lemma *matcheq-matchNone*: $\neg$ *has-primitive m* $\Longrightarrow$ *matcheq-matchNone m* $\longleftrightarrow$ $\neg$ *matches* γ *m p*
⟨*proof*⟩

lemma *matcheq-matchNone-not-matches*: *matcheq-matchNone m* $\Longrightarrow$ $\neg$ *matches* γ *m p*
⟨*proof*⟩

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

lemma *matches-rule-iptables-bigstep*:
assumes *matches* γ *m p* $\longleftrightarrow$ *matches* γ *m′ p*
shows $\Gamma,\gamma,p\vdash$ ⟨[*Rule m a*], *s*⟩ $\Rightarrow$ *t* $\longleftrightarrow$ $\Gamma,\gamma,p\vdash$ ⟨[*Rule m′ a*], *s*⟩ $\Rightarrow$ *t* (**is** *?l* $\longleftrightarrow$ *?r*)
⟨*proof*⟩

lemma *matches-rule-and-simp-help*:
assumes *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash$ ⟨[*Rule* (*MatchAnd m m′*) *a′*], *Undecided*⟩ $\Rightarrow$ *t* $\longleftrightarrow$ $\Gamma,\gamma,p\vdash$ ⟨[*Rule m′ a′*], *Undecided*⟩ $\Rightarrow$ *t* (**is** *?l* $\longleftrightarrow$ *?r*)
⟨*proof*⟩

lemma *matches-MatchNot-simp*:
assumes *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash$ ⟨[*Rule* (*MatchNot m*) *a*], *Undecided*⟩ $\Rightarrow$ *t* $\longleftrightarrow$ $\Gamma,\gamma,p\vdash$ ⟨[], *Undecided*⟩ $\Rightarrow$ *t* (**is** *?l* $\longleftrightarrow$ *?r*)
⟨*proof*⟩

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*)
$\langle$*proof*$\rangle$

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*
$\langle$*proof*$\rangle$

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*
$\langle$*proof*$\rangle$

4.2 Add match

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

lemma *add-match-split*: *add-match m* (*rs1*@*rs2*) = *add-match m rs1* @ *add-match m rs2*
$\langle$*proof*$\rangle$

lemma *add-match-split-fst*: *add-match m* (*Rule m' a'* # *rs*) = *Rule* (*MatchAnd m m'*) *a'* # *add-match m rs*
$\langle$*proof*$\rangle$

lemma *add-match-distrib*:
$\Gamma,\gamma,p\vdash$ $\langle$*add-match m1* (*add-match m2 rs*), *s*$\rangle$ $\Rightarrow$ *t* $\longleftrightarrow$ $\Gamma,\gamma,p\vdash$ $\langle$*add-match m2* (*add-match m1 rs*), *s*$\rangle$ $\Rightarrow$ *t*
$\langle$*proof*$\rangle$

lemma *add-match-split-fst'*: *add-match m* (*a* # *rs*) = *add-match m* [*a*] @ *add-match m rs*
$\langle$*proof*$\rangle$

lemma *matches-add-match-simp*:
assumes *m*: *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash$ $\langle$*add-match m rs*, *s*$\rangle$ $\Rightarrow$ *t* $\longleftrightarrow$ $\Gamma,\gamma,p\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow$ *t* (**is** *?l* $\longleftrightarrow$ *?r*)
$\langle$*proof*$\rangle$

lemma *matches-add-match-MatchNot-simp*:

assumes m: *matches* γ m p
shows $\Gamma,\gamma,p\vdash \langle$*add-match* (*MatchNot* m) *rs*, $s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle[], s\rangle \Rightarrow t$ (**is** *?l s* $\longleftrightarrow$ *?r s*)
$\langle$*proof*$\rangle$

lemma *not-matches-add-match-simp*:
assumes $\neg$ *matches* γ m p
shows $\Gamma,\gamma,p\vdash \langle$*add-match m rs*, *Undecided*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle[]$, *Undecided*$\rangle \Rightarrow t$
$\langle$*proof*$\rangle$

lemma *iptables-bigstep-add-match-notnot-simp*:
$\Gamma,\gamma,p\vdash \langle$*add-match* (*MatchNot* (*MatchNot* m)) *rs*, $s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle$*add-match* m *rs*, $s\rangle \Rightarrow t$
$\langle$*proof*$\rangle$

lemma *add-match-match-not-cases*:
$\Gamma,\gamma,p\vdash \langle$*add-match* (*MatchNot* m) *rs*, *Undecided*$\rangle \Rightarrow$ *Undecided* $\Longrightarrow$ *matches* γ m p $\vee$ $\Gamma,\gamma,p\vdash \langle$*rs*, *Undecided*$\rangle \Rightarrow$ *Undecided*
$\langle$*proof*$\rangle$

lemma *not-matches-add-matchNot-simp*:
$\neg$ *matches* γ m p $\Longrightarrow$ $\Gamma,\gamma,p\vdash \langle$*add-match* (*MatchNot* m) *rs*, $s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle$*rs*, $s\rangle \Rightarrow t$
$\langle$*proof*$\rangle$

lemma *iptables-bigstep-add-match-and*:
$\Gamma,\gamma,p\vdash \langle$*add-match m1* (*add-match m2 rs*), $s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle$*add-match* (*MatchAnd m1 m2*) *rs*, $s\rangle \Rightarrow t$
$\langle$*proof*$\rangle$

lemma *optimize-matches-option-generic*:
assumes $\forall$ $r \in$ *set rs*. P (*get-match* r)
and ($\bigwedge m\ m'$. $P\ m \Longrightarrow f\ m = $ *Some* $m' \Longrightarrow$ *matches* γ m' p = *matches* γ m p)
and ($\bigwedge m$. $P\ m \Longrightarrow f\ m = $ *None* $\Longrightarrow \neg$ *matches* γ m p)
shows $\Gamma,\gamma,p\vdash \langle$*optimize-matches-option f rs*, $s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle$*rs*, $s\rangle \Rightarrow t$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
$\langle$*proof*$\rangle$

lemma *optimize-matches-generic*: $\forall$ $r \in$ *set rs*. P (*get-match* r) $\Longrightarrow$
($\bigwedge m$. $P\ m \Longrightarrow$ *matches* γ (f m) p = *matches* γ m p) $\Longrightarrow$
$\Gamma,\gamma,p\vdash \langle$*optimize-matches f rs*, $s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle$*rs*, $s\rangle \Rightarrow t$
$\langle$*proof*$\rangle$
end
theory *Ruleset-Update*
imports *Matching*

begin

lemma *free-return-not-match*: $\Gamma,\gamma,p\vdash \langle[Rule\ m\ Return],\ Undecided\rangle \Rightarrow t \Longrightarrow \neg$ $matches\ \gamma\ m\ p$
$\langle proof \rangle$

4.3 Background Ruleset Updating

lemma *update-Gamma-nomatch*:
assumes $\neg\ matches\ \gamma\ m\ p$
shows $\Gamma(chain \mapsto Rule\ m\ a\ \#\ rs),\gamma,p\vdash \langle rs',\ s\rangle \Rightarrow t \longleftrightarrow \Gamma(chain \mapsto rs),\gamma,p\vdash$ $\langle rs',\ s\rangle \Rightarrow t$ (**is** $?l \longleftrightarrow\ ?r$)
$\langle proof \rangle$

lemma *update-Gamma-log-empty*:
assumes $a = Log \lor a = Empty$
shows $\Gamma(chain \mapsto Rule\ m\ a\ \#\ rs),\gamma,p\vdash \langle rs',\ s\rangle \Rightarrow t \longleftrightarrow$
$\Gamma(chain \mapsto rs),\gamma,p\vdash \langle rs',\ s\rangle \Rightarrow t$ (**is** $?l \longleftrightarrow\ ?r$)
$\langle proof \rangle$

lemma *map-update-chain-if*: $(\lambda b.\ if\ b = chain\ then\ Some\ rs\ else\ \Gamma\ b) = \Gamma(chain$ $\mapsto rs)$
$\langle proof \rangle$

lemma *no-recursive-calls-helper*:
assumes $\Gamma,\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow t$
and $matches\ \gamma\ m\ p$
and $\Gamma\ chain = Some\ [Rule\ m\ (Call\ chain)]$
shows $False$
$\langle proof \rangle$

lemma *no-recursive-calls*:
$\Gamma(chain \mapsto [Rule\ m\ (Call\ chain)]),\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow t$ $\Longrightarrow matches\ \gamma\ m\ p \Longrightarrow False$
$\langle proof \rangle$

lemma *no-recursive-calls2*:
assumes $\Gamma(chain \mapsto (Rule\ m\ (Call\ chain))\ \#\ rs''),\gamma,p\vdash \langle(Rule\ m\ (Call\ chain))$ $\#\ rs',\ Undecided\rangle \Rightarrow Undecided$
and $matches\ \gamma\ m\ p$
shows $False$
$\langle proof \rangle$

lemma *update-Gamma-nochange1*:
assumes $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle[Rule\ m\ a],\ Undecided\rangle \Rightarrow Undecided$
and $\Gamma(chain \mapsto Rule\ m\ a\ \#\ rs),\gamma,p\vdash \langle rs',\ s\rangle \Rightarrow t$
shows $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle rs',\ s\rangle \Rightarrow t$
$\langle proof \rangle$

lemma *update-gamme-remove-Undecidedpart*:
assumes $\Gamma(chain \mapsto rs')$,$\gamma$,$p\vdash \langle rs', Undecided\rangle \Rightarrow Undecided$
and $\Gamma(chain \mapsto rs1@rs')$,$\gamma$,$p\vdash \langle rs, Undecided\rangle \Rightarrow Undecided$
shows $\Gamma(chain \mapsto rs')$,$\gamma$,$p\vdash \langle rs, Undecided\rangle \Rightarrow Undecided$
⟨*proof*⟩

lemma *update-Gamma-nocall*:
assumes $\neg (\exists chain.\ a = Call\ chain)$
shows Γ,γ,$p\vdash \langle [Rule\ m\ a], s\rangle \Rightarrow t \longleftrightarrow \Gamma'$,$\gamma$,$p\vdash \langle [Rule\ m\ a], s\rangle \Rightarrow t$
⟨*proof*⟩

lemma *update-Gamma-call*:
assumes $\Gamma\ chain = Some\ rs$ **and** $\Gamma'\ chain = Some\ rs'$
assumes Γ,γ,$p\vdash \langle rs, Undecided\rangle \Rightarrow Undecided$ **and** Γ',γ,$p\vdash \langle rs', Undecided\rangle \Rightarrow Undecided$
shows Γ,γ,$p\vdash \langle [Rule\ m\ (Call\ chain)], s\rangle \Rightarrow t \longleftrightarrow \Gamma'$,$\gamma$,$p\vdash \langle [Rule\ m\ (Call\ chain)], s\rangle \Rightarrow t$
⟨*proof*⟩

lemma *update-Gamma-remove-call-undecided*:
assumes $\Gamma(chain \mapsto Rule\ m\ (Call\ foo)\ \#\ rs')$,$\gamma$,$p\vdash \langle rs, Undecided\rangle \Rightarrow Undecided$
and $matches\ \gamma\ m\ p$
shows $\Gamma(chain \mapsto rs')$,$\gamma$,$p\vdash \langle rs, Undecided\rangle \Rightarrow Undecided$
⟨*proof*⟩

lemma *all-return-subchain*:
assumes *a1*: $\Gamma\ chain = Some\ rs$
and *a2*: $matches\ \gamma\ m\ p$
and *a3*: $\forall r\in set\ rs.\ get\text{-}action\ r = Return$
shows Γ,γ,$p\vdash \langle [Rule\ m\ (Call\ chain)], Undecided\rangle \Rightarrow Undecided$
⟨*proof*⟩

lemma *get-action-case-simp*: $get\text{-}action\ (case\ r\ of\ Rule\ m'\ x \Rightarrow Rule\ (MatchAnd\ m\ m')\ x) = get\text{-}action\ r$
⟨*proof*⟩

lemma *updategamma-insert-new*: Γ,γ,$p\vdash \langle rs, s\rangle \Rightarrow t \Longrightarrow chain \notin dom\ \Gamma \Longrightarrow \Gamma(chain \mapsto rs')$,$\gamma$,$p\vdash \langle rs, s\rangle \Rightarrow t$
⟨*proof*⟩

end
theory *Call-Return-Unfolding*
imports *Matching Ruleset-Update*

Common/Repeat-Stabilize
begin

5 *Call Return* Unfolding

Remove *Returns*

fun *process-ret* :: *'a rule list* ⇒ *'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 *Calls*

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*)
⟨*proof*⟩

lemma *process-ret-split-fst-NeqReturn*:
a ≠ *Return* ⟹ *process-ret*((*Rule m a*) # *rs*) = (*Rule m a*) # (*process-ret rs*)
⟨*proof*⟩

lemma *add-match-simp*: *add-match m* = *map* (λ*r*. *Rule* (*MatchAnd m* (*get-match r*)) (*get-action r*))
⟨*proof*⟩

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*⟩

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

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*⟩

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*
⟨*proof*⟩

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))
⟨*proof*⟩

lemma *add-missing-ret-unfoldings-emptyrs2*: *add-missing-ret-unfoldings rs1* [] = []
⟨*proof*⟩

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

lemma *process-call-split-fst*: *process-call* Γ (*a* # *rs*) = *process-call* Γ [*a*] @ *process-call* Γ *rs*
⟨*proof*⟩

lemma *iptables-bigstep-process-ret-undecided*: Γ,γ,p⊢ ⟨*rs*, *Undecided*⟩ ⇒ *t* ⟹ Γ,γ,p⊢ ⟨*process-ret rs*, *Undecided*⟩ ⇒ *t*
⟨*proof*⟩

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*

⟨*proof*⟩

Completeness

theorem *unfolding-complete*: $\Gamma,\gamma,p\vdash \langle rs,s\rangle \Rightarrow t \implies \Gamma,\gamma,p\vdash \langle \textit{process-call}\ \Gamma\ rs,s\rangle \Rightarrow t$
⟨*proof*⟩

lemma *process-ret-cases*:
$\textit{process-ret}\ rs = rs \vee (\exists rs_1\ rs_2\ m.\ rs = rs_1 @[\textit{Rule}\ m\ \textit{Return}]@rs_2 \wedge (\textit{process-ret}\ rs) = rs_1@(\textit{process-ret}\ ([\textit{Rule}\ m\ \textit{Return}]@rs_2)))$
⟨*proof*⟩

lemma *process-ret-splitcases*:
obtains (*id*) $\textit{process-ret}\ rs = rs$
| (*split*) $rs_1\ rs_2\ m$ **where** $rs = rs_1@[\textit{Rule}\ m\ \textit{Return}]@rs_2$ **and** $\textit{process-ret}\ rs = rs_1@(\textit{process-ret}\ ([\textit{Rule}\ m\ \textit{Return}]@rs_2))$
⟨*proof*⟩

lemma *iptables-bigstep-process-ret-cases3*:
assumes $\Gamma,\gamma,p\vdash \langle \textit{process-ret}\ rs,\ \textit{Undecided}\rangle \Rightarrow \textit{Undecided}$
obtains (*noreturn*) $\Gamma,\gamma,p\vdash \langle rs,\ \textit{Undecided}\rangle \Rightarrow \textit{Undecided}$
| (*return*) $rs_1\ rs_2\ m$ **where** $rs = rs_1@[\textit{Rule}\ m\ \textit{Return}]@rs_2$ $\Gamma,\gamma,p\vdash \langle rs_1,\ \textit{Undecided}\rangle \Rightarrow \textit{Undecided}$ $\textit{matches}\ \gamma\ m\ p$
⟨*proof*⟩

lemma *iptables-bigstep-process-ret-DecisionD*: $\Gamma,\gamma,p\vdash \langle \textit{process-ret}\ rs,\ s\rangle \Rightarrow \textit{Decision}\ X \implies \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow \textit{Decision}\ X$
⟨*proof*⟩

5.2 *process-ret* correctness

lemma *process-ret-add-match-dist1*: $\Gamma,\gamma,p\vdash \langle \textit{process-ret}\ (\textit{add-match}\ m\ rs),\ s\rangle \Rightarrow t \implies \Gamma,\gamma,p\vdash \langle \textit{add-match}\ m\ (\textit{process-ret}\ rs),\ s\rangle \Rightarrow t$
⟨*proof*⟩

lemma *process-ret-add-match-dist2*: $\Gamma,\gamma,p\vdash \langle \textit{add-match}\ m\ (\textit{process-ret}\ rs),\ s\rangle \Rightarrow t \implies \Gamma,\gamma,p\vdash \langle \textit{process-ret}\ (\textit{add-match}\ m\ rs),\ s\rangle \Rightarrow t$
⟨*proof*⟩

lemma *process-ret-add-match-dist*: $\Gamma,\gamma,p\vdash \langle \textit{process-ret}\ (\textit{add-match}\ m\ rs),\ s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle \textit{add-match}\ m\ (\textit{process-ret}\ rs),\ s\rangle \Rightarrow t$
⟨*proof*⟩

lemma *process-ret-Undecided-sound*:

assumes $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle process\text{-}ret\ (add\text{-}match\ m\ rs),\ Undecided\rangle \Rightarrow Undecided$
shows $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow Undecided$
⟨proof⟩

lemma *process-ret-Decision-sound*:
assumes $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle process\text{-}ret\ (add\text{-}match\ m\ rs),\ Undecided\rangle \Rightarrow Decision\ X$
shows $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow Decision\ X$
⟨proof⟩

lemma *process-ret-result-empty*: $[] = process\text{-}ret\ rs \Longrightarrow \forall\, r \in set\ rs.\ get\text{-}action\ r = Return$
⟨proof⟩

lemma *process-ret-sound'*:
assumes $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle process\text{-}ret\ (add\text{-}match\ m\ rs),\ Undecided\rangle \Rightarrow t$
shows $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow t$
⟨proof⟩

lemma *wf-chain-process-ret*: $wf\text{-}chain\ \Gamma\ rs \Longrightarrow wf\text{-}chain\ \Gamma\ (process\text{-}ret\ rs)$
⟨proof⟩
lemma *wf-chain-add-match*: $wf\text{-}chain\ \Gamma\ rs \Longrightarrow wf\text{-}chain\ \Gamma\ (add\text{-}match\ m\ rs)$
⟨proof⟩

5.3 Soundness

theorem *unfolding-sound*: $wf\text{-}chain\ \Gamma\ rs \Longrightarrow \Gamma,\gamma,p\vdash \langle process\text{-}call\ \Gamma\ rs,\ s\rangle \Rightarrow t \Longrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
⟨proof⟩

corollary *unfolding-sound-complete*: $wf\text{-}chain\ \Gamma\ rs \Longrightarrow \Gamma,\gamma,p\vdash \langle process\text{-}call\ \Gamma\ rs,\ s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
⟨proof⟩

corollary *unfolding-n-sound-complete*: $\forall\, rsg \in ran\ \Gamma \cup \{rs\}.\ wf\text{-}chain\ \Gamma\ rsg \Longrightarrow \Gamma,\gamma,p\vdash \langle((process\text{-}call\ \Gamma)^{\frown\frown}n)\ rs,\ s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
⟨proof⟩

```
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* $[''X'' \mapsto [Rule\ (Match\ b)\ Return,\ Rule\ (Match\ c)\ Accept]]\ [Rule\ (Match\ a)\ (Call\ ''X'')] =$
$[Rule\ (MatchAnd\ (Match\ a)\ (MatchAnd\ (MatchNot\ (Match\ b))\ (Match\ c)))\ Accept]$ ⟨*proof*⟩

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** $chain\text{-}name \in set\ (map\ fst\ \Gamma)$ **and** $default\text{-}action = Accept \vee default\text{-}action = Drop$
shows $(map\text{-}of\ \Gamma),\gamma,p\vdash \langle repeat\text{-}stabilize\ n\ (process\text{-}call\ (map\text{-}of\ \Gamma))\ [Rule\ MatchAny\ (Call\ chain\text{-}name),\ Rule\ MatchAny\ default\text{-}action],\ s\rangle \Rightarrow t \longleftrightarrow$
$(map\text{-}of\ \Gamma),\gamma,p\vdash \langle [Rule\ MatchAny\ (Call\ chain\text{-}name),\ Rule\ MatchAny\ default\text{-}action],\ s\rangle \Rightarrow t$
⟨*proof*⟩

definition *unfold-optimize-ruleset-CHAIN*
:: $('a\ match\text{-}expr \Rightarrow 'a\ match\text{-}expr) \Rightarrow string \Rightarrow action \Rightarrow 'a\ ruleset \Rightarrow 'a\ rule\ list\ option$
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** $chain\text{-}name \in set\ (map\ fst\ \Gamma)$
and $default\text{-}action = Accept \vee default\text{-}action = Drop$
and $\bigwedge m.\ matches\ \gamma\ (optimize\ m)\ p = matches\ \gamma\ m\ p$
and $unfold\text{-}optimize\text{-}ruleset\text{-}CHAIN\ optimize\ chain\text{-}name\ default\text{-}action\ (map\text{-}of\ \Gamma) = Some\ rs$
shows $(map\text{-}of\ \Gamma),\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t \longleftrightarrow$
$(map\text{-}of\ \Gamma),\gamma,p\vdash \langle [Rule\ MatchAny\ (Call\ chain\text{-}name),\ Rule\ MatchAny\ default\text{-}action],\ s\rangle \Rightarrow t$
⟨*proof*⟩

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*
⟨*proof*⟩
lemma *ternary-to-bool-bool-to-ternary*: *ternary-to-bool* (*bool-to-ternary X*) = *Some X*
⟨*proof*⟩
lemma *ternary-to-bool-None*: *ternary-to-bool t* = *None* ⟷ *t* = *TernaryUnknown*
⟨*proof*⟩
lemma *ternary-to-bool-SomeE*: *ternary-to-bool t* = *Some X* ⟹
(*t* = *TernaryTrue* ⟹ *X* = *True* ⟹ *P*) ⟹ (*t* = *TernaryFalse* ⟹ *X* = *False* ⟹ *P*) ⟹ *P*
⟨*proof*⟩
lemma *ternary-to-bool-Some*: *ternary-to-bool t* = *Some X* ⟷
(*t* = *TernaryTrue* ∧ *X* = *True*) ∨ (*t* = *TernaryFalse* ∧ *X* = *False*)
⟨*proof*⟩
lemma *bool-to-ternary-Unknown*: *bool-to-ternary t* = *TernaryUnknown* ⟷ *False*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *eval-ternary-Not-UnknownD*: *eval-ternary-Not t* = *TernaryUnknown* ⟹ *t* = *TernaryUnknown*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *eval-ternary-idempotence-Not*: *eval-ternary-Not* (*eval-ternary-Not a*) = *a*
⟨*proof*⟩

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*
⟨*proof*⟩

context
begin
private lemma *bool-to-ternary-simp1*: *bool-to-ternary X* = *TernaryTrue* ⟷ *X*
⟨*proof*⟩ **lemma** *bool-to-ternary-simp2*: *bool-to-ternary Y* = *TernaryFalse* ⟷ ¬ *Y*
⟨*proof*⟩ **lemma** *bool-to-ternary-simp3*: *eval-ternary-Not* (*bool-to-ternary X*) = *TernaryTrue* ⟷ ¬ *X*
⟨*proof*⟩ **lemma** *bool-to-ternary-simp4*: *eval-ternary-Not* (*bool-to-ternary X*) = *TernaryFalse* ⟷ *X*
⟨*proof*⟩ **lemma** *bool-to-ternary-simp5*: ¬ (*eval-ternary-Not* (*bool-to-ternary X*) = *TernaryUnknown*)
⟨*proof*⟩ **lemma** *bool-to-ternary-simp6*: *bool-to-ternary X* ≠ *TernaryUnknown*
⟨*proof*⟩

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*)
⟨*proof*⟩ **lemma** *bool-to-ternary-pullup2*:
eval-ternary-And (*bool-to-ternary X1*) (*bool-to-ternary X2*) = *bool-to-ternary* (*X1* ∧ *X2*)
⟨*proof*⟩ **lemma** *bool-to-ternary-pullup3*:
eval-ternary-Imp (*bool-to-ternary X1*) (*bool-to-ternary X2*) = *bool-to-ternary* (*X1* ⟶ *X2*)
⟨*proof*⟩ **lemma** *bool-to-ternary-pullup4*:
eval-ternary-Or (*bool-to-ternary X1*) (*bool-to-ternary X2*) = *bool-to-ternary* (*X1* ∨ *X2*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *ternary-ternary-eval-idempotence-Not*:
ternary-ternary-eval (*TernaryNot* (*TernaryNot a*)) = *ternary-ternary-eval a*
⟨*proof*⟩

lemma *ternary-ternary-eval-TernaryAnd-comm*:
ternary-ternary-eval (*TernaryAnd t1 t2*) = *ternary-ternary-eval* (*TernaryAnd t2 t1*)
⟨*proof*⟩

lemma *eval-ternary-Not* (*ternary-ternary-eval t*) = (*ternary-ternary-eval* (*TernaryNot t*)) ⟨*proof*⟩

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*
⟨*proof*⟩ **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*
⟨*proof*⟩

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*⟩

lemma *NNF-ternary-NegationNormalForm*: *NegationNormalForm* (*NNF-ternary t*)
⟨*proof*⟩

context
begin
private lemma *ternary-lift1*: *eval-ternary-Not tv* ≠ *TernaryFalse* ⟷ *tv* = *TernaryFalse* ∨ *tv* = *TernaryUnknown*
⟨*proof*⟩ **lemma** *ternary-lift2*: *eval-ternary-Not tv* ≠ *TernaryTrue* ⟷ *tv* = *TernaryTrue* ∨ *tv* = *TernaryUnknown*

⟨*proof*⟩ **lemma** *ternary-lift3*: *eval-ternary-Not tv* = *TernaryFalse* ⟷ *tv* = *TernaryTrue*
⟨*proof*⟩ **lemma** *ternary-lift4*: *eval-ternary-Not tv* = *TernaryTrue* ⟷ *tv* = *TernaryFalse*
⟨*proof*⟩ **lemma** *ternary-lift5*: *eval-ternary-Not tv* = *TernaryUnknown* ⟷ *tv* = *TernaryUnknown*
⟨*proof*⟩ **lemma** *ternary-lift6*: *eval-ternary-And t1 t2* = *TernaryFalse* ⟷ *t1* = *TernaryFalse* ∨ *t2* = *TernaryFalse*
⟨*proof*⟩ **lemma** *ternary-lift7*: *eval-ternary-And t1 t2* = *TernaryTrue* ⟷ *t1* = *TernaryTrue* ∧ *t2* = *TernaryTrue*
⟨*proof*⟩

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* ⟹ *tv* = *TernaryFalse*
⟨*proof*⟩ **lemma** *l2*: *eval-ternary-And t1 t2* = *TernaryFalse* ⟹ *t1* = *TernaryFalse* ∨ *t2* = *TernaryFalse*
⟨*proof*⟩

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*))
⟨*proof*⟩
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*))

⟨*proof*⟩

lemma *matches-case*: *matches* γ *m a p* $\longleftrightarrow$ (*case ternary-eval* (*map-match-tac* (*fst* γ) *p m*) *of None* $\Rightarrow$ (*snd* γ) *a p* | *Some b* $\Rightarrow$ *b*)
⟨*proof*⟩

lemma *matches-case-tuple*: *matches* (β, α) *m a p* $\longleftrightarrow$ (*case ternary-eval* (*map-match-tac* β *p m*) *of None* $\Rightarrow$ α *a p* | *Some b* $\Rightarrow$ *b*)
⟨*proof*⟩

lemma *matches-case-ternaryvalue-tuple*: *matches* (β, α) *m a p* $\longleftrightarrow$ (*case ternary-ternary-eval* (*map-match-tac* β *p m*) *of*
TernaryUnknown $\Rightarrow$ α *a p* |
TernaryTrue $\Rightarrow$ *True* |
TernaryFalse $\Rightarrow$ *False*)
⟨*proof*⟩

lemma *matches-casesE*:
matches (β, α) *m a p* $\Longrightarrow$
(*ternary-ternary-eval* (*map-match-tac* β *p m*) = *TernaryUnknown* $\Longrightarrow$ α *a p* $\Longrightarrow$ *P*) $\Longrightarrow$
(*ternary-ternary-eval* (*map-match-tac* β *p m*) = *TernaryTrue* $\Longrightarrow$ *P*)
$\Longrightarrow$ *P*
⟨*proof*⟩

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

lemma ⟦ *ternary-ternary-eval* (*map-match-tac* β *p expr*) = *TernaryUnknown* ⟧ $\Longrightarrow$ *matches* (β, α) *expr a p* $\longleftrightarrow$ *matches* (β, α) (*MatchNot expr*) *a p*
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩
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*))
⟨*proof*⟩

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*)
$\langle proof \rangle$

7.1 Ternary Matcher Algebra

lemma *matches-and-comm*: *matches* γ (*MatchAnd m m'*) *a p* $\longleftrightarrow$ *matches* γ (*MatchAnd m' m*) *a p*
$\langle proof \rangle$

lemma *matches-not-idem*: *matches* γ (*MatchNot* (*MatchNot m*)) *a p* $\longleftrightarrow$ *matches* γ *m a p*
$\langle proof \rangle$

lemma *MatchOr*: *matches* γ (*MatchOr m1 m2*) *a p* $\longleftrightarrow$ *matches* γ *m1 a p* $\vee$ *matches* γ *m2 a p*
$\langle proof \rangle$

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*
$\langle proof \rangle$

lemma (*TernaryNot* (*map-match-tac* β *p* (*m*))) = (*map-match-tac* β *p* (*MatchNot m*))
$\langle proof \rangle$

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*
$\langle proof \rangle$ **lemma** *matches-simp11*: *matches* γ *m a p* $\Longrightarrow$ *matches* γ (*MatchAnd m' m*) *a p* $\longleftrightarrow$ *matches* γ *m' a p*
$\langle proof \rangle$ **lemma** *matches-simp2*: *matches* γ (*MatchAnd m m'*) *a p* $\Longrightarrow$ $\neg$ *matches* γ *m a p* $\Longrightarrow$ *False*
$\langle proof \rangle$ **lemma** *matches-simp22*: *matches* γ (*MatchAnd m m'*) *a p* $\Longrightarrow$ $\neg$ *matches* γ *m' a p* $\Longrightarrow$ *False*
$\langle proof \rangle$ **lemma** *matches-simp3*: *matches* γ (*MatchNot m*) *a p* $\Longrightarrow$ *matches* γ *m a p* $\Longrightarrow$ (*snd* γ) *a p*
$\langle proof \rangle$ **lemma** *matches* γ (*MatchNot m*) *a p* $\Longrightarrow$ *matches* γ *m a p* $\Longrightarrow$ (*ternary-eval* (*map-match-tac* (*fst* γ) *p m*)) = *None*
$\langle proof \rangle$

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *opt-MatchAny-match-expr-correct*: *matches* γ (*opt-MatchAny-match-expr* *m*) = *matches* γ *m*
⟨*proof*⟩

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 α ≡ (α *Drop* = α *Reject*)

lemma *wf-unknown-match-tacD-False1*: *wf-unknown-match-tac* α $\Longrightarrow$ ¬ *matches* (β, α) *m Reject p* $\Longrightarrow$ *matches* (β, α) *m Drop p* $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *wf-unknown-match-tacD-False2*: *wf-unknown-match-tac* α $\Longrightarrow$ *matches* (β, α) *m Reject p* $\Longrightarrow$ ¬ *matches* (β, α) *m Drop p* $\Longrightarrow$ *False*
⟨*proof*⟩

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*) |

— ¬ $(a \wedge b)$ = ¬ *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′*)))
)
⟨*proof*⟩

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*
($\forall$ *p*. *ternary-ternary-eval* (*map-match-tac* β *p* (*Match A*)) = *TernaryUnknown*)
then
if ($\forall$ *p*. α *a p*) *then MatchAny else if* ($\forall$ *p*. $\neg$ α *a p*) *then MatchNot MatchAny else MatchNot* (*Match A*)
else MatchNot (*Match A*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *remove-unknowns-generic*: *matches* γ (*remove-unknowns-generic* γ *a m*) *a* = *matches* γ *m a*
⟨*proof*⟩

fun *has-unknowns* :: (*'a*, *'p*) *exact-match-tac* ⇒ *'a match-expr* ⇒ *bool* **where**
has-unknowns β (*Match A*) = ($\exists$ *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* $\vee$ *has-unknowns* β *m2*)

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

lemma *packet-independent-unknown-match*: *a* = *Accept* $\vee$ *a* = *Drop* ⟹ *packet-independent-*α α ⟹ $\neg$ *unknown-not-match-any* α *a* ⟷ *unknown-match-all* α *a*
⟨*proof*⟩

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 $\beta \equiv \forall A. (\exists p. \beta\ A\ p \neq$ *TernaryUnknown*$) \longrightarrow$ $(\forall p. \beta\ A\ p \neq$ *TernaryUnknown*$)$

lemma *remove-unknowns-generic-specification*: $a = Accept \lor a = Drop \Longrightarrow$ *packet-independent-α* $\alpha \Longrightarrow$
packet-independent-β-unknown $\beta \Longrightarrow$
$\neg$ *has-unknowns* β (*remove-unknowns-generic* (β, α) a m)
⟨*proof*⟩

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*⟩ **lemma** *no-primitives-matchNot*: **assumes** ¬ *has-primitive m* **shows** *matches* γ (*MatchNot m*) a p ⟷ ¬ *matches* γ m a p
⟨*proof*⟩

lemma *matcheq-matchAny*: ¬ *has-primitive m* ⟹ *matcheq-matchAny m* ⟷ *matches* γ m a p
⟨*proof*⟩

lemma *matcheq-matchNone*: ¬ *has-primitive m* ⟹ *matcheq-matchNone m* ⟷ ¬ *matches* γ m a p
⟨*proof*⟩

lemma *matcheq-matchNone-not-matches*: *matcheq-matchNone m* ⟹ ¬ *matches* γ m a p
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

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* $\neq$ α *Accept* **and** *packet-independent-*α α
  and $\forall$ *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*⟩

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* ⇒ *'p* ⇒ *'a rule list* ⇒ *state* ⇒ *state* ⇒ *bool*
  (‹-,-⊢ ⟨-, -⟩ $\Rightarrow_\alpha$ -› [*60*,*60*,*20*,*98*,*98*] *89*)
  for γ **and** p **where**
skip: γ,p⊢ ⟨[], *t*⟩ $\Rightarrow_\alpha$ *t* |
accept: ⟦*matches* γ *m Accept p*⟧ ⟹ γ,p⊢ ⟨[*Rule m Accept*], *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow* |
drop: ⟦*matches* γ *m Drop p*⟧ ⟹ γ,p⊢ ⟨[*Rule m Drop*], *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny* |
reject: ⟦*matches* γ *m Reject p*⟧ ⟹ γ,p⊢ ⟨[*Rule m Reject*], *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny* |
log: ⟦*matches* γ *m Log p*⟧ ⟹ γ,p⊢ ⟨[*Rule m Log*], *Undecided*⟩ $\Rightarrow_\alpha$ *Undecided* |
empty: ⟦*matches* γ *m Empty p*⟧ ⟹ γ,p⊢ ⟨[*Rule m Empty*], *Undecided*⟩ $\Rightarrow_\alpha$ *Undecided* |
nomatch: ⟦¬ *matches* γ *m a p*⟧ ⟹ γ,p⊢ ⟨[*Rule m a*], *Undecided*⟩ $\Rightarrow_\alpha$ *Undecided* |
decision: γ,p⊢ ⟨*rs*, *Decision X*⟩ $\Rightarrow_\alpha$ *Decision X* |
seq: ⟦γ,p⊢ ⟨rs_1, *Undecided*⟩ $\Rightarrow_\alpha$ *t*; γ,p⊢ ⟨rs_2, *t*⟩ $\Rightarrow_\alpha$ t'⟧ ⟹ γ,p⊢ ⟨rs_1@rs_2, *Undecided*⟩ $\Rightarrow_\alpha$ 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*] : γ,p⊢ ⟨*rs*,*s*⟩ $\Rightarrow_\alpha$ *t* ⟹

$(\bigwedge t.\ P\ []\ t\ t) \Longrightarrow$
$(\bigwedge m\ a.\ matches\ \gamma\ m\ a\ p \Longrightarrow a = Accept \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalAllow)) \Longrightarrow$
$(\bigwedge m\ a.\ matches\ \gamma\ m\ a\ p \Longrightarrow a = Drop \vee a = Reject \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalDeny)) \Longrightarrow$
$(\bigwedge m\ a.\ matches\ \gamma\ m\ a\ p \Longrightarrow a = Log \vee a = Empty \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided) \Longrightarrow$
$(\bigwedge m\ a.\ \neg\ matches\ \gamma\ m\ a\ p \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided) \Longrightarrow$
$(\bigwedge rs\ X.\ P\ rs\ (Decision\ X)\ (Decision\ X)) \Longrightarrow$
$(\bigwedge rs\ rs_1\ rs_2\ t\ t'.\ rs = rs_1\ @\ rs_2 \Longrightarrow \gamma,p\vdash \langle rs_1, Undecided\rangle \Rightarrow_\alpha t \Longrightarrow P\ rs_1\ Undecided\ t \Longrightarrow \gamma,p\vdash \langle rs_2, t\rangle \Rightarrow_\alpha t' \Longrightarrow P\ rs_2\ t\ t' \Longrightarrow P\ rs\ Undecided\ t')$
$\Longrightarrow P\ rs\ s\ t$
⟨*proof*⟩

lemma *skipD*: $\gamma,p\vdash \langle [],\ s\rangle \Rightarrow_\alpha t \Longrightarrow s = t$
⟨*proof*⟩

lemma *decisionD*: $\gamma,p\vdash \langle rs,\ Decision\ X\rangle \Rightarrow_\alpha t \Longrightarrow t = Decision\ X$
⟨*proof*⟩

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*⟩

lemma *dropD*: $\gamma,p\vdash \langle [Rule\ m\ Drop],\ Undecided\rangle \Rightarrow_\alpha t \Longrightarrow matches\ \gamma\ m\ Drop\ p \Longrightarrow t = Decision\ FinalDeny$
⟨*proof*⟩

lemma *rejectD*: $\gamma,p\vdash \langle [Rule\ m\ Reject],\ Undecided\rangle \Rightarrow_\alpha t \Longrightarrow matches\ \gamma\ m\ Reject\ p \Longrightarrow t = Decision\ FinalDeny$
⟨*proof*⟩

lemma *logD*: $\gamma,p\vdash \langle [Rule\ m\ Log],\ Undecided\rangle \Rightarrow_\alpha t \Longrightarrow t = Undecided$
⟨*proof*⟩

lemma *emptyD*: $\gamma,p\vdash \langle [Rule\ m\ Empty],\ Undecided\rangle \Rightarrow_\alpha t \Longrightarrow t = Undecided$
⟨*proof*⟩

lemma *nomatchD*: $\gamma,p\vdash \langle [Rule\ m\ a],\ Undecided\rangle \Rightarrow_\alpha t \Longrightarrow \neg\ matches\ \gamma\ m\ a\ p \Longrightarrow t = Undecided$
⟨*proof*⟩

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

lemma *approximating-bigstep-to-undecided*: $\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha Undecided \Longrightarrow s = Undecided$
⟨*proof*⟩

lemma *approximating-bigstep-to-decision1*: $\gamma,p\vdash \langle rs,\ Decision\ Y\rangle \Rightarrow_\alpha Decision\ X \Longrightarrow Y = X$
⟨*proof*⟩

lemma *nomatch-fst*: $\neg$ *matches* γ *m a p* $\Longrightarrow$ $\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t \Longrightarrow \gamma,p\vdash \langle Rule\ m\ a\ \#\ rs,\ s\rangle \Rightarrow_\alpha t$
⟨*proof*⟩

lemma *seq'*:
assumes $rs = rs_1\ @\ rs_2\ \gamma,p\vdash \langle rs_1,s\rangle \Rightarrow_\alpha t\ \gamma,p\vdash \langle rs_2,t\rangle \Rightarrow_\alpha t'$
shows $\gamma,p\vdash \langle rs,s\rangle \Rightarrow_\alpha t'$
⟨*proof*⟩

lemma *seq-split*:
assumes $\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t\ rs = rs_1@rs_2$
obtains t' **where** $\gamma,p\vdash \langle rs_1,s\rangle \Rightarrow_\alpha t'\ \gamma,p\vdash \langle rs_2,t'\rangle \Rightarrow_\alpha t$
⟨*proof*⟩

lemma *seqE-fst*:
assumes $\gamma,p\vdash \langle r\#rs,\ s\rangle \Rightarrow_\alpha t$
obtains t' **where** $\gamma,p\vdash \langle [r],s\rangle \Rightarrow_\alpha t'\ \gamma,p\vdash \langle rs,t'\rangle \Rightarrow_\alpha t$
⟨*proof*⟩

lemma *seq-fst*: **assumes** $\gamma,p\vdash \langle [r],\ s\rangle \Rightarrow_\alpha t$ **and** $\gamma,p\vdash \langle rs,\ t\rangle \Rightarrow_\alpha t'$ **shows** $\gamma,p\vdash \langle r\ \#\ rs,\ s\rangle \Rightarrow_\alpha t'$
⟨*proof*⟩

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*
⟨*proof*⟩
lemma *wf-rulesetD*: **assumes** *wf-ruleset* γ *p* (*r* # *rs*) **shows** *wf-ruleset* γ *p* [*r*] **and** *wf-ruleset* γ *p rs*
⟨*proof*⟩

lemma *wf-ruleset-fst*: *wf-ruleset* γ *p* (*Rule m a* # *rs*) $\longleftrightarrow$ *wf-ruleset* γ *p* [*Rule m a*] $\wedge$ *wf-ruleset* γ *p rs*
⟨*proof*⟩
lemma *wf-ruleset-stripfst*: *wf-ruleset* γ *p* (*r* # *rs*) $\Longrightarrow$ *wf-ruleset* γ *p* (*rs*)
⟨*proof*⟩
lemma *wf-ruleset-rest*: *wf-ruleset* γ *p* (*Rule m a* # *rs*) $\Longrightarrow$ *wf-ruleset* γ *p* [*Rule m a*]
⟨*proof*⟩

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* γ *p* [] *s*) $\Longrightarrow$
($\bigwedge\gamma$ *p r rs X*. *P* γ *p* (*r* # *rs*) (*Decision X*)) $\Longrightarrow$
($\bigwedge\gamma$ *p m a rs*.
¬ *matches* γ *m a p* $\Longrightarrow$ *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* = *Log* $\Longrightarrow$ *P* γ *p rs Undecided*) $\Longrightarrow$ (*a* = *Empty* $\Longrightarrow$ *P* γ *p rs Undecided*) $\Longrightarrow$ *P* γ *p* (*Rule m a* # *rs*) *Undecided*) $\Longrightarrow$
P γ *p rs s*
⟨*proof*⟩

lemma *Decision-approximating-bigstep-fun*: *approximating-bigstep-fun* γ *p rs* (*Decision X*) = *Decision X*
⟨*proof*⟩

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

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*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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

lemma *approximating-bigstep-wf*: $\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow_\alpha Undecided \Longrightarrow$ *wf-ruleset* γ *p rs*
⟨*proof*⟩

only valid actions appear in this ruleset

definition *good-ruleset* :: *'a rule list* $\Rightarrow$ *bool* **where**
good-ruleset rs $\equiv$ $\forall$ *r* $\in$ *set rs*. ($\neg$($\exists$ *chain*. *get-action r* = *Call chain*) $\wedge$ *get-action r* $\neq$ *Return* $\wedge$ $\neg$($\exists$ *chain*. *get-action r* = *Goto chain*) $\wedge$ *get-action r* $\neq$ *Unknown*)

lemma[*code*]: *good-ruleset rs* = ($\forall$ *r*$\in$*set rs*. (*case get-action r of Call chain* $\Rightarrow$ *False* | *Return* $\Rightarrow$ *False* | *Goto chain* $\Rightarrow$ *False* | *Unknown* $\Rightarrow$ *False* | - $\Rightarrow$ *True*))
⟨*proof*⟩

lemma *good-ruleset-alt*: *good-ruleset rs* = ($\forall$ *r*$\in$*set rs*. *get-action r* = *Accept* $\vee$ *get-action r* = *Drop* $\vee$
get-action r = *Reject* $\vee$ *get-action r* = *Log* $\vee$ *get-action r* = *Empty*)
⟨*proof*⟩

lemma *good-ruleset-append*: *good-ruleset* (rs_1 @ rs_2) $\longleftrightarrow$ *good-ruleset* rs_1 $\wedge$ *good-ruleset* rs_2

⟨*proof*⟩

lemma *good-ruleset-fst*: *good-ruleset* $(r\#rs) \Longrightarrow$ *good-ruleset* $[r]$
⟨*proof*⟩
lemma *good-ruleset-tail*: *good-ruleset* $(r\#rs) \Longrightarrow$ *good-ruleset rs*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *simple-imp-good-ruleset*: *simple-ruleset rs* $\Longrightarrow$ *good-ruleset rs*
⟨*proof*⟩

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*⟩

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*⟩

lemma *approximating-fun-imp-semantics*: **assumes** *wf-ruleset* γ *p rs*
shows *approximating-bigstep-fun* γ *p rs s* = *t* $\Longrightarrow \gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t$
⟨*proof*⟩

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* $\Longrightarrow$
$\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t \longleftrightarrow$ *approximating-bigstep-fun* γ *p rs s* = *t*
⟨*proof*⟩

corollary *approximating-semantics-iff-fun-good-ruleset*: *good-ruleset rs* $\Longrightarrow$
$\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t \longleftrightarrow$ *approximating-bigstep-fun* γ *p rs s* = *t*
⟨*proof*⟩

lemma *approximating-bigstep-deterministic*: $[\![$ $\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t$; $\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t'$ $]\!] \Longrightarrow t = t'$
⟨*proof*⟩

lemma *rm-LogEmpty-fun-semantics*:
approximating-bigstep-fun γ *p* (*rm-LogEmpty rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
⟨*proof*⟩

lemma $\gamma,p\vdash \langle$*rm-LogEmpty rs*, $s\rangle \Rightarrow_\alpha t \longleftrightarrow \gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t$
⟨*proof*⟩

lemma *rm-LogEmpty-simple-but-Reject*:
good-ruleset rs $\Longrightarrow \forall r \in$ *set* (*rm-LogEmpty rs*). *get-action r* = *Accept* ∨ *get-action r* = *Reject* ∨ *get-action r* = *Drop*
⟨*proof*⟩

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*⟩

lemma *rmLogEmpty-rwReject-good-to-simple*: *good-ruleset rs* $\Longrightarrow$ *simple-ruleset* (*rw-Reject* (*rm-LogEmpty rs*))
⟨*proof*⟩

8.5 Matching

lemma *optimize-matches-option-generic*:
assumes ∀ *r* ∈ *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* $\Longrightarrow$ ¬ *matches* γ *m a p*)
shows *approximating-bigstep-fun* γ *p* (*optimize-matches-option f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
⟨*proof*⟩

lemma *optimize-matches-generic*: ∀ *r* ∈ *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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *optimize-matches*: ∀ *m a*. *matches* γ (*f m*) *a p* = *matches* γ *m a p* $\Longrightarrow$ *ap-*

proximating-bigstep-fun γ p (optimize-matches f rs) s = approximating-bigstep-fun γ p rs s
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *optimize-matches-a*: *∀ a m. matches γ m a = matches γ (f a m) a ⟹ approximating-bigstep-fun γ p (optimize-matches-a f rs) s = approximating-bigstep-fun γ p rs s*
⟨*proof*⟩

lemma *optimize-matches-a-simplers*:
assumes *simple-ruleset rs* **and** *∀ a m. a = Accept ∨ a = Drop ⟶ 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*⟩

lemma *not-matches-removeAll*: *¬ matches γ m a p ⟹*
approximating-bigstep-fun γ p (removeAll (Rule m a) rs) Undecided = approximating-bigstep-fun γ p rs Undecided
⟨*proof*⟩

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 ⇒ bool) × ('a ⇒ 'b)) ⇒ ('b ⇒ 'a) ⇒ bool* **where**
wf-disc-sel (disc, sel) C ⟷ (∀ a. disc a ⟶ C (sel a) = a) ∧ (∀ a. ~~disc (C a) ⟶~~ 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*}
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

end

lemma *ipt-iprange-to-set-uncurry-IpAddrNetmask*:
ipt-iprange-to-set (*uncurry IpAddrNetmask a*) = *uncurry ipset-from-cidr a*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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

lemma *wordinterval-to-set* (*l2wi* (*map ipt-iprange-to-interval ips*)) = ($\bigcup$ *ip* ∈ *set ips*. *ipt-iprange-to-set ip*)
⟨*proof*⟩

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*) =
($\bigcap$ *ip* ∈ *set* (*getPos l*). *ipt-iprange-to-set ip*) − ($\bigcup$ *ip* ∈ *set* (*getNeg l*). *ipt-iprange-to-set ip*)
⟨*proof*⟩

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*:
($\bigcup$ *ip* ∈ *set* (*wi-2-cidr-ipt-iprange-list r*). *ipt-iprange-to-set ip*) = *wordinterval-to-set r*
⟨*proof*⟩

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*: ($\bigcup$ *ip* ∈ *set* (*ipt-iprange-compress l*). *ipt-iprange-to-set ip*) =
($\bigcap$ *ip* ∈ *set* (*getPos l*). *ipt-iprange-to-set ip*) − ($\bigcup$ *ip* ∈ *set* (*getNeg l*). *ipt-iprange-to-set ip*)
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *ipt-iprange-compress-normalized-IpAddrNetmask*:
∀ *a'*∈*set* (*ipt-iprange-compress as*). *normalized-cidr-ip a'*

⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*} ⟨*proof*⟩

lemma *match-tcp-flags-nomatch*: ¬ *c* ⊆ *fmask* ⟹ ¬ *match-tcp-flags* (*TCP-Flags fmask c*) *pkt* ⟨*proof*⟩

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* ⟨*proof*⟩

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* ⟨*proof*⟩

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

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*
⟨*proof*⟩
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* ⟨*proof*⟩

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*
⟨*proof*⟩
lemma *match-tcp-flags-conjunct-option-None*: *match-tcp-flags-conjunct-option f1*

f2 = *None* $\Longrightarrow$
¬(*match-tcp-flags f1 pkt* ∧ *match-tcp-flags f2 pkt*)
⟨*proof*⟩

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*
⟨*proof*⟩

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* $\Longrightarrow$ (*c1* = *c2* ∧ *fmask1* = *fmask2*) = (∀ *pkt*. (*pkt* ∩ *fmask1* = *c1*) = (*pkt* ∩ *fmask2* = *c2*))
⟨*proof*⟩

lemma *ipt-tcp-flags-equal*: *ipt-tcp-flags-equal f1 f2* ⟷ (∀ *pkt*. *match-tcp-flags f1 pkt* = *match-tcp-flags f2 pkt*)
⟨*proof*⟩
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*⟩

We can reuse the wordinterval theory to reason about ports

lemma *ports-to-set-wordinterval*: *ports-to-set ps* = *wordinterval-to-set* (*l2wi ps*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *UNIV-ctstate*: *UNIV* = {*CT-New*, *CT-Established*, *CT-Related*, *CT-Untracked*, *CT-Invalid*} ⟨*proof*⟩

instance *ctstate* :: *finite*
⟨*proof*⟩

lemma *finite* (*S* :: *ctstate set*) ⟨*proof*⟩

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*⟩
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*
⟨*proof*⟩

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''*
⟨*proof*⟩

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*⦈
⟨*proof*⟩

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
  ⟨proof⟩
  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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *packet-independent-unknown-match-tacs*:
packet-independent-α *in-doubt-allow*
packet-independent-α *in-doubt-deny*
⟨*proof*⟩

lemma *Drop-neq-Accept-unknown-match-tacs*:
in-doubt-allow Drop $\neq$ *in-doubt-allow Accept*

in-doubt-deny Drop $\neq$ *in-doubt-deny Accept*
⟨*proof*⟩

corollary *matches-induction-case-MatchNot-in-doubt-allow*:
$\forall$ *a. matches* (β,*in-doubt-allow*) *m′ a p* = *matches* (β,*in-doubt-allow*) *m a p* $\Longrightarrow$
matches (β,*in-doubt-allow*) (*MatchNot m′*) *a p* = *matches* (β,*in-doubt-allow*) (*MatchNot m*) *a p*
⟨*proof*⟩
corollary *matches-induction-case-MatchNot-in-doubt-deny*:
$\forall$ *a. matches* (β,*in-doubt-deny*) *m′ a p* = *matches* (β,*in-doubt-deny*) *m a p* $\Longrightarrow$
matches (β,*in-doubt-deny*) (*MatchNot m′*) *a p* = *matches* (β,*in-doubt-deny*) (*MatchNot m*) *a p*
⟨*proof*⟩

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*: $\forall$ *p i.* β (*IIface i*) *p* = *bool-to-ternary* (*match-iface i* (*p-iiface p*))
and *OIface*: $\forall$ *p i.* β (*OIface i*) *p* = *bool-to-ternary* (*match-iface i* (*p-oiface p*))
and *Prot*: $\forall$ *p proto.* β (*Prot proto*) *p* = *bool-to-ternary* (*match-proto proto* (*p-proto p*))
and *Src-Ports*: $\forall$ *p proto ps.* β (*Src-Ports* (*L4Ports proto ps*)) *p* = *bool-to-ternary* (*proto* = *p-proto p* $\wedge$ *p-sport p* $\in$ *ports-to-set ps*)
and *Dst-Ports*: $\forall$ *p proto ps.* β (*Dst-Ports* (*L4Ports proto ps*)) *p* = *bool-to-ternary* (*proto* = *p-proto p* $\wedge$ *p-dport p* $\in$ *ports-to-set ps*)
— -m multiport –ports matches source or destination port
and *MultiportsPorts*: $\forall$ *p proto ps.* β (*MultiportPorts* (*L4Ports proto ps*)) *p* = *bool-to-ternary* (*proto* = *p-proto p* $\wedge$ (*p-sport p* $\in$ *ports-to-set ps* $\vee$ *p-dport p* $\in$ *ports-to-set ps*))
and *L4-Flags*: $\forall$ *p flags.* β (*L4-Flags flags*) *p* = *bool-to-ternary* (*match-tcp-flags flags* (*p-tcp-flags p*))
and *CT-State*: $\forall$ *p S.* β (*CT-State S*) *p* = *bool-to-ternary* (*match-ctstate S* (*p-tag-ctstate p*))

and *Extra*: $\forall$ *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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Prot-single*:
matches (β, α) (*Match* (*Prot X*)) *a p* $\longleftrightarrow$ *match-proto X* (*p-proto p*)
⟨*proof*⟩
lemma *Prot-single-not*:
matches (β, α) (*MatchNot* (*Match* (*Prot X*))) *a p* $\longleftrightarrow$ $\neg$ *match-proto X* (*p-proto p*)
⟨*proof*⟩

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*
⟨*proof*⟩
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*
⟨*proof*⟩

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* $\longleftrightarrow$
matches (β, α) (*MatchNot* (*Match* (*Prot* (*Proto proto*)))) *a p* $\vee$ *p-dport p* $\notin$ *ports-to-set ps*

⟨*proof*⟩

lemma *multiports-disjuction*:
($\exists$ *rg*$\in$*set spts. matches* (β, α) (*Match* (*Src-Ports* (*L4Ports proto* [*rg*]))) *a p*) $\longleftrightarrow$ *matches* (β, α) (*Match* (*Src-Ports* (*L4Ports proto spts*))) *a p*
($\exists$ *rg*$\in$*set dpts. matches* (β, α) (*Match* (*Dst-Ports* (*L4Ports proto* [*rg*]))) *a p*) $\longleftrightarrow$ *matches* (β, α) (*Match* (*Dst-Ports* (*L4Ports proto dpts*))) *a p*
⟨*proof*⟩

lemma *MultiportPorts-single-rewrite*:
matches (β, α) (*Match* (*MultiportPorts ports*)) *a p* $\longleftrightarrow$
matches (β, α) (*Match* (*Src-Ports ports*)) *a p* $\vee$ *matches* (β, α) (*Match* (*Dst-Ports ports*)) *a p*
⟨*proof*⟩
lemma *MultiportPorts-single-rewrite-MatchOr*:
matches (β, α) (*Match* (*MultiportPorts ports*)) *a p* $\longleftrightarrow$
matches (β, α) (*MatchOr* (*Match* (*Src-Ports ports*)) (*Match* (*Dst-Ports ports*))) *a p*
⟨*proof*⟩

lemma *MultiportPorts-single-not-rewrite-MatchAnd*:
matches (β, α) (*MatchNot* (*Match* (*MultiportPorts ports*))) *a p* $\longleftrightarrow$
matches (β, α) (*MatchAnd* (*MatchNot* (*Match* (*Src-Ports ports*))) (*MatchNot* (*Match* (*Dst-Ports ports*)))) *a p*
⟨*proof*⟩
lemma *MultiportPorts-single-not-rewrite*:
matches (β, α) (*MatchNot* (*Match* (*MultiportPorts ports*))) *a p* $\longleftrightarrow$
$\neg$ *matches* (β, α) (*Match* (*Src-Ports ports*)) *a p* $\wedge$ $\neg$ *matches* (β, α) (*Match* (*Dst-Ports ports*)) *a p*
⟨*proof*⟩

lemma *Extra-single*:
matches (β, α) (*Match* (*Extra str*)) *a p* $\longleftrightarrow$ α *a p*
⟨*proof*⟩
lemma *Extra-single-not*: — ternary logic, $\neg$ *unknown* = *unknown*
matches (β, α) (*MatchNot* (*Match* (*Extra str*))) *a p* $\longleftrightarrow$ α *a p*
⟨*proof*⟩
end

14.2 Basic optimisations

Compress many *Extra* expressions to one expression.

fun *compress-extra* :: *'i::len common-primitive match-expr* $\Rightarrow$ *'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*⟩

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*
⟨*proof*⟩

lemma *primitive-matcher-generic-common-matcher*: *primitive-matcher-generic common-matcher*
⟨*proof*⟩

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*|)
⟨*proof*⟩

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*|)
⟨*proof*⟩

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*⦈
⟨*proof*⟩

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*
⟨*proof*⟩
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*
⟨*proof*⟩

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*
⟨*proof*⟩
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
⟨*proof*⟩
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* ∈ (⋂ *x*∈*set X*. *ipt-iprange-to-set x*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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 ∨ a = Drop ⟹ *remove-unknowns-generic (common-matcher, in-doubt-allow) a m = upper-closure-matchexpr a m*
⟨*proof*⟩

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* ∨ *a* = *Drop* ⟹ *remove-unknowns-generic* (*common-matcher*, *in-doubt-deny*) *a m* = *lower-closure-matchexpr a m*
⟨*proof*⟩

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* ⟨*proof*⟩
lemma[*simp*]: *Semantics.matches applies-Yes* (*Match e*) *p* ⟨*proof*⟩

definition *m*=*Match* (*Src* (*IpAddr* (*0*::*ipv4addr*)))
lemma[*simp*]: *Semantics.matches applies-Yes m p* ⟨*proof*⟩

lemma [″*FORWARD*″ ↦ [(*Rule m Log*), (*Rule m Accept*), (*Rule m Drop*)]],*applies-Yes*,*p*⊢
⟨[*Rule MatchAny* (*Call* ″*FORWARD*″)], *Undecided*⟩ ⇒ (*Decision FinalAllow*)

⟨*proof*⟩

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*)
⟨*proof*⟩

lemma [''*FORWARD*'' ↦ [*Rule m* (*Call ''foo''*), *Rule m Drop*], ''*foo*'' ↦ []],*applies-Yes*,*p*⊢
⟨[*Rule MatchAny* (*Call ''FORWARD''*)], *Undecided*⟩ ⇒ (*Decision FinalDeny*)
⟨*proof*⟩

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*] ⟨*proof*⟩

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*)
⟨*proof*⟩

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*
(‹-,-,-⊢ ⟨-, -⟩ $\Rightarrow_s$ -› [*60*,*60*,*60*,*20*,*98*,*98*] *89*)
for Γ **and** γ **and** *p* **where**
skip: Γ,γ,*p*⊢ ⟨[], *t*⟩ $\Rightarrow_s$ *t* |
accept: *matches* γ *m p* ⟹ Γ,γ,*p*⊢ ⟨*Rule m Accept* # *rs*, *Undecided*⟩ $\Rightarrow_s$ *Decision FinalAllow* |
drop: *matches* γ *m p* ⟹ Γ,γ,*p*⊢ ⟨*Rule m Drop* # *rs*, *Undecided*⟩ $\Rightarrow_s$ *Decision FinalDeny* |
reject: *matches* γ *m p* ⟹ Γ,γ,*p*⊢ ⟨*Rule m Reject* # *rs*, *Undecided*⟩ $\Rightarrow_s$ *Decision FinalDeny* |
log: *matches* γ *m p* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_s$ *t* ⟹ Γ,γ,*p*⊢ ⟨*Rule m Log* # *rs*, *Undecided*⟩ $\Rightarrow_s$ *t* |
empty: *matches* γ *m p* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_s$ *t* ⟹ Γ,γ,*p*⊢ ⟨*Rule m Empty* # *rs*, *Undecided*⟩ $\Rightarrow_s$ *t* |
nms: ¬ *matches* γ *m p* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_s$ *t* ⟹ Γ,γ,*p*⊢ ⟨*Rule m a* # *rs*, *Undecided*⟩ $\Rightarrow_s$ *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*⟩ $\Rightarrow_s$ *Undecided*; Γ,γ,*p*⊢ ⟨*rrs*, *Undecided*⟩ $\Rightarrow_s$ *t* ⟧ ⟹
Γ,γ,*p*⊢ ⟨*Rule m* (*Call chain*) # *rrs*, *Undecided*⟩ $\Rightarrow_s$ *t* |
call-result: ⟦ *matches* γ *m p*; Γ *chain* = *Some rs*; Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_s$ *Decision X* ⟧ ⟹
Γ,γ,*p*⊢ ⟨*Rule m* (*Call chain*) # *rrs*, *Undecided*⟩ $\Rightarrow_s$ *Decision X* |
call-no-result: ⟦ *matches* γ *m p*; Γ *chain* = *Some rs*; Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_s$ *Undecided*;
Γ,γ,*p*⊢ ⟨*rrs*, *Undecided*⟩ $\Rightarrow_s$ *t* ⟧ ⟹
Γ,γ,*p*⊢ ⟨*Rule m* (*Call chain*) # *rrs*, *Undecided*⟩ $\Rightarrow_s$ *t*

private lemma *a*: Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_s$ *t* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t*
⟨*proof*⟩ **lemma** *empty-rs-stateD*: **assumes** Γ,γ,*p*⊢ ⟨[], *s*⟩ $\Rightarrow_s$ *t* **shows** *t* = *s*
⟨*proof*⟩ **lemma** *decided*: ⟦Γ,γ,*p*⊢ ⟨rs_1, *Undecided*⟩ $\Rightarrow_s$ *Decision X*⟧ ⟹ Γ,γ,*p*⊢ ⟨rs_1@rs_2, *Undecided*⟩ $\Rightarrow_s$ *Decision X*
⟨*proof*⟩ **lemma** *decided-determ*: ⟦Γ,γ,*p*⊢ ⟨rs_1, *s*⟩ $\Rightarrow_s$ *t*; *s* = *Decision X*⟧ ⟹ *t* = *Decision X*
⟨*proof*⟩ **lemma** *seq-ns*:
⟦Γ,γ,*p*⊢ ⟨rs_1, *Undecided*⟩ $\Rightarrow_s$ *t*; Γ,γ,*p*⊢ ⟨rs_2, *t*⟩ $\Rightarrow_s$ *t'*⟧ ⟹ Γ,γ,*p*⊢ ⟨rs_1@rs_2, *Undecided*⟩ $\Rightarrow_s$ *t'*
⟨*proof*⟩ **lemma** *b*: Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t* ⟹ *s* = *Undecided* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_s$ *t*
⟨*proof*⟩ **inductive** *iptables-bigstep-nz* :: *'a ruleset* ⇒ (*'a*, *'p*) *matcher* ⇒ *'p* ⇒ *'a rule list* ⇒ *state* ⇒ *bool*
(‹-,-,-⊢ - $\Rightarrow_z$ -› [*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* $\Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m Accept # rs* $\Rightarrow_z$ *Decision FinalAllow* |
drop: *matches* γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m Drop # rs* $\Rightarrow_z$ *Decision FinalDeny* |
reject: *matches* γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m Reject # rs* $\Rightarrow_z$ *Decision FinalDeny* |
log: *matches* γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_z$ *t* $\Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m Log # rs* $\Rightarrow_z$ *t* |
empty: *matches* γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_z$ *t* $\Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m Empty # rs* $\Rightarrow_z$ *t* |
nms: $\neg$ *matches* γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_z$ *t* $\Longrightarrow \Gamma,\gamma,p\vdash$ *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*; $\Gamma,\gamma,p\vdash$ $rs_1 \Rightarrow_z$ *Undecided*; $\Gamma,\gamma,p\vdash$ *rrs* $\Rightarrow_z$ *t* $]\!] \Longrightarrow$
$\Gamma,\gamma,p\vdash$ *Rule m* (*Call chain*) *# rrs* $\Rightarrow_z$ *t* |
call-result: $[\![$ *matches* γ *m p*; Γ *chain* = *Some rs*; $\Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_z$ *Decision X* $]\!] \Longrightarrow$
$\Gamma,\gamma,p\vdash$ *Rule m* (*Call chain*) *# rrs* $\Rightarrow_z$ *Decision X* |
call-no-result: $[\![$ *matches* γ *m p*; Γ *chain* = *Some rs*; $\Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_z$ *Undecided*;
$\Gamma,\gamma,p\vdash$ *rrs* $\Rightarrow_z$ *t* $]\!] \Longrightarrow$
$\Gamma,\gamma,p\vdash$ *Rule m* (*Call chain*) *# rrs* $\Rightarrow_z$ *t*

private lemma *c*: $\Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_z$ *t* $\Longrightarrow \Gamma,\gamma,p\vdash \langle$*rs, Undecided*$\rangle \Rightarrow_s$ *t*
$\langle$*proof*$\rangle$ **lemma** *d*: $\Gamma,\gamma,p\vdash \langle$*rs, s*$\rangle \Rightarrow_s$ *t* $\Longrightarrow$ *s* = *Undecided* $\Longrightarrow \Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_z$ *t*
$\langle$*proof*$\rangle$

inductive *iptables-bigstep-r* :: *′a ruleset* $\Rightarrow$ (*′a, ′p*) *matcher* $\Rightarrow$ *′p* $\Rightarrow$ *′a rule list* $\Rightarrow$ *state* $\Rightarrow$ *bool*
(‹-,-,-⊢ - $\Rightarrow_r$ -› [*60,60,60,20,98*] *89*)
for Γ **and** γ **and** *p* **where**
skip: $\Gamma,\gamma,p \vdash [] \ \Rightarrow_r$ *Undecided* |
accept: *matches* γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m Accept # rs* $\Rightarrow_r$ *Decision FinalAllow* |
drop: *matches* γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m Drop # rs* $\Rightarrow_r$ *Decision FinalDeny* |
reject: *matches* γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m Reject # rs* $\Rightarrow_r$ *Decision FinalDeny* |
return: *matches* γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m Return # rs* $\Rightarrow_r$ *Undecided* |
log: $\Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_r$ *t* $\Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m Log # rs* $\Rightarrow_r$ *t* |
empty: $\Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_r$ *t* $\Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m Empty # rs* $\Rightarrow_r$ *t* |
nms: $\neg$ *matches* γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_r$ *t* $\Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m a # rs* $\Rightarrow_r$ *t* |
call-result: $[\![$ *matches* γ *m p*; Γ *chain* = *Some rs*; $\Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_r$ *Decision X* $]\!] \Longrightarrow$
$\Gamma,\gamma,p\vdash$ *Rule m* (*Call chain*) *# rrs* $\Rightarrow_r$ *Decision X* |
call-no-result: $[\![$ Γ *chain* = *Some rs*; $\Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_r$ *Undecided*;
$\Gamma,\gamma,p\vdash$ *rrs* $\Rightarrow_r$ *t* $]\!] \Longrightarrow$
$\Gamma,\gamma,p\vdash$ *Rule m* (*Call chain*) *# rrs* $\Rightarrow_r$ *t*

private lemma *returning*: $[\![\Gamma,\gamma,p\vdash$ $rs_1 \Rightarrow_r$ *Undecided*; *matches* γ *m′ p*$]\!]$
$\Longrightarrow \Gamma,\gamma,p\vdash$ rs_1 @ *Rule m′ Return #* $rs_2 \Rightarrow_r$ *Undecided*
$\langle$*proof*$\rangle$ **lemma** *e*: $\Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_z$ *t* $\Longrightarrow$ *s* = *Undecided* $\Longrightarrow \Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_r$ *t*
$\langle$*proof*$\rangle$

definition *no-call-to c rs* $\equiv$ ($\forall$ *r* $\in$ *set rs. case get-action r of Call c′* $\Rightarrow$ *c* $\neq$ *c′* | - $\Rightarrow$ *True*)
definition *all-chains p* Γ *rs* $\equiv$ (*p rs* $\wedge$ ($\forall$ *l rs.* Γ *l* = *Some rs* $\longrightarrow$ *p rs*))
private lemma *all-chains-no-call-upd*: *all-chains* (*no-call-to c*) Γ *rs* $\Longrightarrow$ (Γ(*c* $\mapsto$

$x)),\gamma,p\vdash rs \Rightarrow_z t \longleftrightarrow \Gamma,\gamma,p\vdash rs \Rightarrow_z t$
⟨*proof*⟩

lemma *updated-call*: $\Gamma(c \mapsto rs),\gamma,p\vdash rs \Rightarrow_z t \Longrightarrow$ *matches* γ *m p* $\Longrightarrow \Gamma(c \mapsto rs),\gamma,p\vdash$ [*Rule m* (*Call c*)] $\Rightarrow_z t$
⟨*proof*⟩ **lemma shows**
log-nz: $\Gamma,\gamma,p\vdash rs \Rightarrow_z t \Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m Log* # $rs \Rightarrow_z t$
and *empty-nz*: $\Gamma,\gamma,p\vdash rs \Rightarrow_z t \Longrightarrow \Gamma,\gamma,p\vdash$ *Rule m Empty* # $rs \Rightarrow_z t$
⟨*proof*⟩ **lemma** *nz-empty-rs-stateD*: **assumes** $\Gamma,\gamma,p\vdash [] \Rightarrow_z t$ **shows** t = *Undecided*
⟨*proof*⟩ **lemma** *upd-callD*: $\Gamma(c \mapsto rs),\gamma,p\vdash$ [*Rule m* (*Call c*)] $\Rightarrow_z t \Longrightarrow$ *matches* γ *m p*
$\Longrightarrow (\Gamma(c \mapsto rs),\gamma,p\vdash rs \Rightarrow_z t \vee (\exists rs_1\ rs_2\ m'.\ rs = rs_1$ @ *Rule m′ Return* # rs_2 $\wedge$ *matches* γ *m′ p* $\wedge\ \Gamma(c \mapsto rs),\gamma,p\vdash rs_1 \Rightarrow_z$ *Undecided* $\wedge$ t = *Undecided*))
⟨*proof*⟩ **lemma** *partial-fun-upd*: $(f(x \mapsto y))\ x$ = *Some y* ⟨*proof*⟩

lemma *f*: $\Gamma,\gamma,p\vdash rs \Rightarrow_r t \Longrightarrow$ *matches* γ *m p* $\Longrightarrow$ *all-chains* (*no-call-to c*) Γ *rs* $\Longrightarrow$
$(\Gamma(c \mapsto rs)),\gamma,p\vdash$ [*Rule m* (*Call c*)] $\Rightarrow_z t$
⟨*proof*⟩

lemma *r-skip-inv*: $\Gamma,\gamma,p\vdash [] \Rightarrow_r t \Longrightarrow t$ = *Undecided*
⟨*proof*⟩

lemma *r-call-eq*: Γ *c* = *Some rs* $\Longrightarrow$ *matches* γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash$ [*Rule m* (*Call c*)] $\Rightarrow_r t \longleftrightarrow \Gamma,\gamma,p\vdash rs \Rightarrow_r t$

⟨*proof*⟩

lemma *call-eq*: Γ *c* = *Some rs* $\Longrightarrow$ *matches* γ *m p* $\Longrightarrow \forall r \in$ *set rs*. *get-action r* $\neq$ *Return* $\Longrightarrow \Gamma,\gamma,p\vdash$ ⟨[*Rule m* (*Call c*)],*s*⟩ $\Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash$ ⟨*rs*,*s*⟩ $\Rightarrow t$
⟨*proof*⟩

theorem *r-eq-orig*: ⟦*all-chains* (*no-call-to c*) Γ *rs*; Γ *c* = *Some rs*⟧ $\Longrightarrow$
$\Gamma,\gamma,p\vdash rs \Rightarrow_r t \longleftrightarrow \Gamma,\gamma,p\vdash$ ⟨[*Rule MatchAny* (*Call c*)], *Undecided*⟩ $\Rightarrow t$
⟨*proof*⟩

lemma *r-no-call*: $\Gamma,\gamma,p\vdash$ *Rule MatchAny* (*Call c*)#*rs* $\Rightarrow_r t \Longrightarrow \Gamma$ *c* = *None* $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *no-call*: $\Gamma,\gamma,p\vdash$ ⟨*rs*, *s*⟩ $\Rightarrow t \Longrightarrow rs$ = [*Rule MatchAny* (*Call c*)] $\Longrightarrow s$ = *Undecided* $\Longrightarrow \Gamma$ *c* = *None* $\Longrightarrow$ *False*
⟨*proof*⟩ **corollary** *r-eq-orig′*: **assumes** $\forall rs \in$ *ran* Γ. *no-call-to c rs*
shows $\Gamma,\gamma,p\vdash$ [*Rule MatchAny* (*Call c*)] $\Rightarrow_r t \longleftrightarrow \Gamma,\gamma,p\vdash$ ⟨[*Rule MatchAny* (*Call c*)], *Undecided*⟩ $\Rightarrow t$

⟨*proof*⟩

lemma *r-tail*: **assumes** $\Gamma,\gamma,p\vdash$ *rs1* $\Rightarrow_r$ *Decision X* **shows** $\Gamma,\gamma,p\vdash$ *rs1* @ *rs2* $\Rightarrow_r$ *Decision X*
⟨*proof*⟩
lemma *r-seq*: $\Gamma,\gamma,p\vdash$ *rs1* $\Rightarrow_r$ *Undecided* $\Longrightarrow \forall\, r \in$ *set rs1*. $\neg$(*get-action r* = *Return* $\wedge$ *matches* γ (*get-match r*) *p*)
$\Longrightarrow \Gamma,\gamma,p\vdash$ *rs2* $\Rightarrow_r$ *t* $\Longrightarrow \Gamma,\gamma,p\vdash$ *rs1* @ *rs2* $\Rightarrow_r$ *t*
⟨*proof*⟩

lemma *r-appendD*: $\Gamma,\gamma,p\vdash$ *rs1* @ *rs2* $\Rightarrow_r$ *t* $\Longrightarrow \exists\, s.\ \Gamma,\gamma,p\vdash$ *rs1* $\Rightarrow_r$ *s*
⟨*proof*⟩

corollary *iptables-bigstep-r-eq*: **assumes** $\forall$ *rs* $\in$ *ran* Γ. *no-call-to c rs A* = *Accept* $\vee$ *A* = *Drop*
shows $\Gamma,\gamma,p\vdash$ [*Rule MatchAny* (*Call c*), *Rule MatchAny A*] $\Rightarrow_r$ *t* $\longleftrightarrow \Gamma,\gamma,p\vdash$ ⟨[*Rule MatchAny* (*Call c*), *Rule MatchAny A*], *Undecided*⟩ $\Rightarrow$ *t*

⟨*proof*⟩

lemma *ex-no-call*: *finite S* $\Longrightarrow \exists\, c.\ \forall$ (*rs* :: *'a rule list*) $\in$ *S*. *no-call-to c rs*

⟨*proof*⟩ **lemma** *ex-no-call'*: *finite* (*dom* Γ) $\Longrightarrow \exists\, c.\ \Gamma$ *c* = *None* $\wedge$ ($\forall$ (*rs* :: *'a rule list*) $\in$ (*ran* Γ). *no-call-to c rs*)

⟨*proof*⟩

lemma *all-chains-no-call-upd-r*: *all-chains* (*no-call-to c*) Γ *rs* $\Longrightarrow$ (Γ(*c* $\mapsto$ *x*)),γ,*p*$\vdash$ *rs* $\Rightarrow_r$ *t* $\longleftrightarrow \Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_r$ *t*
⟨*proof*⟩

lemma *all-chains-no-call-upd-orig*: *all-chains* (*no-call-to c*) Γ *rs* $\Longrightarrow$ (Γ(*c* $\mapsto$ *x*)),γ,*p*$\vdash$ ⟨*rs*,*s*⟩ $\Rightarrow$ *t* $\longleftrightarrow \Gamma,\gamma,p\vdash$ ⟨*rs*,*s*⟩ $\Rightarrow$ *t*
⟨*proof*⟩

corollary *r-eq-orig'''*: **assumes** *finite* (*ran* Γ) **and** $\forall\, r \in$ *set rs*. *get-action r* $\neq$ *Return*
shows $\Gamma,\gamma,p\vdash$ *rs* $\Rightarrow_r$ *t* $\longleftrightarrow \Gamma,\gamma,p\vdash$ ⟨*rs*, *Undecided*⟩ $\Rightarrow$ *t*
⟨*proof*⟩

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 ⇒
(*'σ* ⇒ (*'a*, *'p*) *matcher*) ⇒ — matcher, first parameter is the state
(*'σ* ⇒ *final-decision* ⇒ *'p* ⇒ *'σ*) ⇒ — state update function after firewall has decision for a packet
'σ ⇒ — Starting state. constant
(*string* × *action*) ⇒ — The chain and default policy the firewall evaluates. For example "FORWARD", Drop
'p list ⇒ — packets to be processed
(*'p* × *final-decision*) *list* ⇒ — packets which have been processed and their decision. ordered the same as the firewall processed them. oldest packet first
'σ ⇒ — 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' ⟹
Γ,(γ_σ σ'),*p*⊢ ⟨[*Rule MatchAny* (*Call built-in-chain*), *Rule MatchAny default-policy*],*Undecided*⟩ ⇒ *Decision X* ⟹
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* ⟹
Γ,γ_σ *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 Γ γ_σ *state-upate* σ_0 (*built-in-chain*, *default-policy*) *ps* *ps-processed* σ'
⟨*proof*⟩

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'
⟨*proof*⟩

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* $\Rightarrow$
($'a$, $'ptagged$) *matcher* $\Rightarrow$
($'\sigma \Rightarrow 'p \Rightarrow 'ptagged$) $\Rightarrow$ — taggs the packet accordig to the current state before processing by firewall
($'\sigma \Rightarrow$ *final-decision* $\Rightarrow 'p \Rightarrow '\sigma$) $\Rightarrow$ — state updater
$'\sigma \Rightarrow$ — Starting state. constant
(*string* $\times$ *action*) $\Rightarrow$
$'p$ *list* $\Rightarrow$ — packets to be processed
($'p \times$ *final-decision*) *list* $\Rightarrow$ — packets which have been processed
$'\sigma \Rightarrow$ — 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* $\sigma' \Longrightarrow$
Γ,γ,(*packet-tagger* σ' *p*)⊢ ⟨[*Rule MatchAny* (*Call built-in-chain*), *Rule MatchAny* *default-policy*],*Undecided*⟩ $\Rightarrow$ *Decision X* $\Longrightarrow$
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* = [] $\Longrightarrow$
semantics-stateful-packet-tagging Γ γ *packet-tagger state-upate* σ_0 (*built-in-chain*, *default-policy*) *ps ps-processed* σ'
⟨*proof*⟩

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* $\Longrightarrow$
Γ,γ,(*packet-tagger* σ*-old p*)⊢ ⟨[*Rule MatchAny* (*Call built-in-chain*), *Rule MatchAny default-policy*], *Undecided*⟩ $\Rightarrow$ *Decision X* $\Longrightarrow$

$\sigma' = \textit{state-upate } \sigma\textit{-old } X\ p \Longrightarrow$
ps-processed = *ps-processed-old*@[(*p*, *X*)] $\Longrightarrow$
semantics-stateful-packet-tagging $\Gamma\ \gamma$ *packet-tagger state-upate* σ_0 (*built-in-chain*, *default-policy*) *ps ps-processed* σ'
⟨*proof*⟩

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*⟩

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*′ 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′
⟨*proof*⟩

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* {})
⟨*proof*⟩

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*)})
⟨*proof*⟩

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*)})
⟨*proof*⟩

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*)})
⟨*proof*⟩

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'
⟨*proof*⟩
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* $\Rightarrow$ *'p* $\Rightarrow$ *'a rule list* $\Rightarrow$ *bool*
where

no-matching-Goto - - [] $\longleftrightarrow$ *True* |
no-matching-Goto γ *p* ((*Rule m* (*Goto* -))#*rs*) $\longleftrightarrow$ $\neg$ *matches* γ *m p* $\wedge$ *no-matching-Goto* γ *p rs* |
no-matching-Goto γ *p* (-#*rs*) $\longleftrightarrow$ *no-matching-Goto* γ *p rs*

inductive *iptables-goto-bigstep* :: *'a ruleset* $\Rightarrow$ $('a, 'p)$ *matcher* $\Rightarrow$ *'p* $\Rightarrow$ *'a rule list* $\Rightarrow$ *state* $\Rightarrow$ *state* $\Rightarrow$ *bool*
$(\langle$-,-,-$\vdash_g$ $\langle$-, -$\rangle$ $\Rightarrow$ -$\rangle$ [60,60,60,20,98,98] 89)
for Γ **and** γ **and** p **where**

skip: $\Gamma,\gamma,p\vdash_g$ $\langle[], t\rangle \Rightarrow t$ |
accept: *matches* γ *m p* $\Longrightarrow$ $\Gamma,\gamma,p\vdash_g$ $\langle$[*Rule m Accept*], *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow* |
drop: *matches* γ *m p* $\Longrightarrow$ $\Gamma,\gamma,p\vdash_g$ $\langle$[*Rule m Drop*], *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalDeny* |
reject: *matches* γ *m p* $\Longrightarrow$ $\Gamma,\gamma,p\vdash_g$ $\langle$[*Rule m Reject*], *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalDeny* |
log: *matches* γ *m p* $\Longrightarrow$ $\Gamma,\gamma,p\vdash_g$ $\langle$[*Rule m Log*], *Undecided*$\rangle$ $\Rightarrow$ *Undecided* |
empty: *matches* γ *m p* $\Longrightarrow$ $\Gamma,\gamma,p\vdash_g$ $\langle$[*Rule m Empty*], *Undecided*$\rangle$ $\Rightarrow$ *Undecided* |
nomatch: $\neg$ *matches* γ *m p* $\Longrightarrow$ $\Gamma,\gamma,p\vdash_g$ $\langle$[*Rule m a*], *Undecided*$\rangle$ $\Rightarrow$ *Undecided* |
decision: $\Gamma,\gamma,p\vdash_g$ $\langle$*rs*, *Decision X*$\rangle$ $\Rightarrow$ *Decision X* |
seq: $[\![\Gamma,\gamma,p\vdash_g \langle rs_1, \textit{Undecided}\rangle \Rightarrow t;\ \Gamma,\gamma,p\vdash_g \langle rs_2, t\rangle \Rightarrow t';$ *no-matching-Goto* $\gamma\ p\ rs_1]\!] \Longrightarrow \Gamma,\gamma,p\vdash_g \langle rs_1@rs_2, \textit{Undecided}\rangle \Rightarrow t'$ |
call-return: $[\![$ *matches* γ *m p*; Γ *chain* = *Some* (rs_1@[*Rule m' Return*]@rs_2);
matches γ *m' p*; $\Gamma,\gamma,p\vdash_g \langle rs_1, \textit{Undecided}\rangle \Rightarrow \textit{Undecided}$;
no-matching-Goto $\gamma\ p\ rs_1]\!] \Longrightarrow$
— 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!
$\Gamma,\gamma,p\vdash_g$ $\langle$[*Rule m* (*Call chain*)], *Undecided*$\rangle$ $\Rightarrow$ *Undecided* |
call-result: $[\![$ *matches* γ *m p*; Γ *chain* = *Some rs*; $\Gamma,\gamma,p\vdash_g \langle rs, \textit{Undecided}\rangle \Rightarrow t$ $]\!] \Longrightarrow$
$\Gamma,\gamma,p\vdash_g$ $\langle$[*Rule m* (*Call chain*)], *Undecided*$\rangle$ $\Rightarrow$ *t* | — goto handling here seems okay
goto-decision: $[\![$ *matches* γ *m p*; Γ *chain* = *Some rs*; $\Gamma,\gamma,p\vdash_g \langle rs, \textit{Undecided}\rangle$ $\Rightarrow$ *Decision X* $]\!] \Longrightarrow$
$\Gamma,\gamma,p\vdash_g$ $\langle$(*Rule m* (*Goto chain*))#*rest*, *Undecided*$\rangle$ $\Rightarrow$ *Decision X* |
goto-no-decision: $[\![$ *matches* γ *m p*; Γ *chain* = *Some rs*; $\Gamma,\gamma,p\vdash_g \langle rs, \textit{Undecided}\rangle$ $\Rightarrow$ *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:

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_1,\ \mathit{Undecided}\rangle \Rightarrow t \\ \Gamma,\gamma,p\vdash_g \langle \mathit{rs}_2,\ t\rangle \Rightarrow t' \qquad \mathit{no\text{-}matching\text{-}Goto}\ \gamma\ p\ \mathit{rs}_1\end{array}}{\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_1\ @\ \mathit{rs}_2,\ \mathit{Undecided}\rangle \Rightarrow t'}$$

$$\frac{\begin{array}{c}\mathit{matches}\ \gamma\ m\ p \\ \Gamma\ \mathit{chain} = \mathit{Some}\ (\mathit{rs}_1\ @\ [\mathit{Rule}\ m'\ \mathit{Return}]\ @\ \mathit{rs}_2) \qquad \mathit{matches}\ \gamma\ m'\ p \\ \Gamma,\gamma,p\vdash_g \langle \mathit{rs}_1,\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided} \qquad \mathit{no\text{-}matching\text{-}Goto}\ \gamma\ p\ \mathit{rs}_1\end{array}}{\Gamma,\gamma,p\vdash_g \langle [\mathit{Rule}\ m\ (\mathit{Call}\ \mathit{chain})],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}}$$

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

$$\frac{\begin{array}{c}\mathit{matches}\ \gamma\ m\ p \\ \Gamma\ \mathit{chain} = \mathit{Some}\ \mathit{rs} \qquad \Gamma,\gamma,p\vdash_g \langle \mathit{rs},\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ X\end{array}}{\Gamma,\gamma,p\vdash_g \langle \mathit{Rule}\ m\ (\mathit{Goto}\ \mathit{chain}) \cdot \mathit{rest},\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ X}$$

$$\frac{\begin{array}{c}\mathit{matches}\ \gamma\ m\ p \\ \Gamma\ \mathit{chain} = \mathit{Some}\ \mathit{rs} \qquad \Gamma,\gamma,p\vdash_g \langle \mathit{rs},\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}\end{array}}{\Gamma,\gamma,p\vdash_g \langle \mathit{Rule}\ m\ (\mathit{Goto}\ \mathit{chain}) \cdot \mathit{rest},\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}}$$

private lemma *deny*:
$\mathit{matches}\ \gamma\ m\ p \Longrightarrow a = \mathit{Drop} \vee a = \mathit{Reject} \Longrightarrow \mathit{iptables\text{-}goto\text{-}bigstep}\ \Gamma\ \gamma\ p$ $[\mathit{Rule}\ m\ a]\ \mathit{Undecided}\ (\mathit{Decision}\ \mathit{FinalDeny})$

⟨*proof*⟩ **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*]:
$[\![\ \Gamma,\gamma,p\vdash_g \langle rs,s\rangle \Rightarrow t;$
$\bigwedge t.\ P\ []\ t\ t;$
$\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Accept \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalAllow);$
$\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Drop \lor a = Reject \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalDeny);$
$\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Log \lor a = Empty \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided;$
$\bigwedge m\ a.\ \neg\ matches\ \gamma\ m\ p \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided;$
$\bigwedge rs\ X.\ P\ rs\ (Decision\ X)\ (Decision\ X);$
$\bigwedge rs\ rs_1\ rs_2\ t\ t'.\ rs = rs_1\ @\ rs_2 \Longrightarrow \Gamma,\gamma,p\vdash_g \langle rs_1,Undecided\rangle \Rightarrow t \Longrightarrow P\ rs_1\ Undecided\ t \Longrightarrow$
$\Gamma,\gamma,p\vdash_g \langle rs_2,t\rangle \Rightarrow t' \Longrightarrow P\ rs_2\ t\ t' \Longrightarrow no\text{-}matching\text{-}Goto\ \gamma\ p\ rs_1 \Longrightarrow$
$P\ rs\ Undecided\ t';$
$\bigwedge m\ a\ chain\ rs_1\ m'\ rs_2.\ matches\ \gamma\ m\ p \Longrightarrow a = Call\ chain \Longrightarrow$
$\Gamma\ chain = Some\ (rs_1\ @\ [Rule\ m'\ Return]\ @\ rs_2) \Longrightarrow$
$matches\ \gamma\ m'\ p \Longrightarrow \Gamma,\gamma,p\vdash_g \langle rs_1,Undecided\rangle \Rightarrow Undecided \Longrightarrow$
$no\text{-}matching\text{-}Goto\ \gamma\ p\ rs_1 \Longrightarrow P\ rs_1\ Undecided\ Undecided \Longrightarrow$
$P\ [Rule\ m\ a]\ Undecided\ Undecided;$
$\bigwedge m\ a\ chain\ rs\ t.\ matches\ \gamma\ m\ p \Longrightarrow a = Call\ chain \Longrightarrow \Gamma\ chain = Some\ rs \Longrightarrow$
$\Gamma,\gamma,p\vdash_g \langle rs,Undecided\rangle \Rightarrow t \Longrightarrow P\ rs\ Undecided\ t \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ t;$
$\bigwedge m\ a\ chain\ rs\ rest\ X.\ matches\ \gamma\ m\ p \Longrightarrow a = Goto\ chain \Longrightarrow \Gamma\ chain = Some\ rs \Longrightarrow$
$\Gamma,\gamma,p\vdash_g \langle rs,Undecided\rangle \Rightarrow (Decision\ X) \Longrightarrow P\ rs\ Undecided\ (Decision\ X) \Longrightarrow$
$P\ (Rule\ m\ a\#rest)\ Undecided\ (Decision\ X);$
$\bigwedge m\ a\ chain\ rs\ rest.\ matches\ \gamma\ m\ p \Longrightarrow a = Goto\ chain \Longrightarrow \Gamma\ chain = Some\ rs \Longrightarrow$
$\Gamma,\gamma,p\vdash_g \langle rs,Undecided\rangle \Rightarrow Undecided \Longrightarrow P\ rs\ Undecided\ Undecided \Longrightarrow$
$P\ (Rule\ m\ a\#rest)\ Undecided\ Undecided]\!] \Longrightarrow$
$P\ rs\ s\ t$
⟨*proof*⟩

17.1.1 Forward reasoning

private lemma *decisionD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow s = Decision\ X \Longrightarrow t = Decision\ X$
⟨*proof*⟩ **lemma** *iptables-goto-bigstep-to-undecided*: $\Gamma,\gamma,p\vdash_g \langle rs,\ s\rangle \Rightarrow Undecided$

$\Longrightarrow$ *s* = *Undecided*

⟨*proof*⟩ **lemma** *iptables-goto-bigstep-to-decision*: $\Gamma,\gamma,p\vdash_g \langle rs,\ Decision\ Y\rangle \Rightarrow Decision\ X \Longrightarrow Y = X$

⟨*proof*⟩ **lemma** *skipD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [] \Longrightarrow s = t$

⟨*proof*⟩ **lemma** *gotoD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ (Goto\ chain)] \Longrightarrow s = Undecided \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow$

$\exists\ rs.\ \Gamma\ chain = Some\ rs \wedge \Gamma,\gamma,p\vdash_g \langle rs,s\rangle \Rightarrow t$

⟨*proof*⟩ **lemma** *not-no-matching-Goto-singleton-cases*: $\neg$ *no-matching-Goto* γ *p* [*Rule m a*] $\longleftrightarrow$ ($\exists$ *chain*. *a* = (*Goto chain*)) $\wedge$ *matches* γ *m p*

⟨*proof*⟩ **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*)

⟨*proof*⟩ **lemma** *no-matching-Goto-head*: *no-matching-Goto* γ *p* (*r*#*rs*) $\Longrightarrow$ *no-matching-Goto* γ *p* [*r*]

⟨*proof*⟩ **lemma** *no-matching-Goto-tail*: *no-matching-Goto* γ *p* (*r*#*rs*) $\Longrightarrow$ *no-matching-Goto* γ *p rs*

⟨*proof*⟩ **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*

⟨*proof*⟩ **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*⟩ **lemma** *seq-cons-Goto-t*:
$\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ (Goto\ chain)],\ Undecided\rangle \Rightarrow t \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow \Gamma,\gamma,p\vdash_g \langle Rule\ m\ (Goto\ chain)\ \#\ rs,\ Undecided\rangle \Rightarrow t$

⟨*proof*⟩ **lemma** *no-matching-Goto-append*: *no-matching-Goto* γ *p* (*rs1*@*rs2*) $\longleftrightarrow$ *no-matching-Goto* γ *p rs1* $\wedge$ *no-matching-Goto* γ *p rs2*

⟨*proof*⟩ **lemma** *no-matching-Goto-append1*: *no-matching-Goto* γ *p* (*rs1*@*rs2*) $\Longrightarrow$ *no-matching-Goto* γ *p rs1*

⟨*proof*⟩ **lemma** *no-matching-Goto-append2*: *no-matching-Goto* γ *p* (*rs1*@*rs2*) $\Longrightarrow$ *no-matching-Goto* γ *p rs2*

⟨*proof*⟩ **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-matching-Goto* γ *p* [*r*]
shows $\Gamma,\gamma,p\vdash_g \langle r\#rs,\ Undecided\rangle \Rightarrow t'$
⟨*proof*⟩

context
notes *skipD*[*dest*] *list-app-singletonE*[*elim*]
begin
lemma *acceptD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Accept] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Decision\ FinalAllow$
⟨*proof*⟩

lemma *dropD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Drop] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Decision\ FinalDeny$
⟨*proof*⟩

lemma *rejectD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [\mathit{Rule}\ m\ \mathit{Reject}] \Longrightarrow \mathit{matches}\ \gamma\ m\ p \Longrightarrow s = \mathit{Undecided} \Longrightarrow t = \mathit{Decision}\ \mathit{FinalDeny}$
⟨*proof*⟩

lemma *logD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [\mathit{Rule}\ m\ \mathit{Log}] \Longrightarrow \mathit{matches}\ \gamma\ m\ p \Longrightarrow s = \mathit{Undecided} \Longrightarrow t = \mathit{Undecided}$
⟨*proof*⟩

lemma *emptyD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [\mathit{Rule}\ m\ \mathit{Empty}] \Longrightarrow \mathit{matches}\ \gamma\ m\ p \Longrightarrow s = \mathit{Undecided} \Longrightarrow t = \mathit{Undecided}$
⟨*proof*⟩

lemma *nomatchD*: $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [\mathit{Rule}\ m\ a] \Longrightarrow s = \mathit{Undecided} \Longrightarrow \neg\ \mathit{matches}\ \gamma\ m\ p \Longrightarrow t = \mathit{Undecided}$
⟨*proof*⟩

lemma *callD*:
assumes $\Gamma,\gamma,p\vdash_g \langle r,\ s\rangle \Rightarrow t$ $r = [\mathit{Rule}\ m\ (\mathit{Call}\ \mathit{chain})]$ $s = \mathit{Undecided}$ $\mathit{matches}\ \gamma\ m\ p$ $\Gamma\ \mathit{chain} = \mathit{Some}\ \mathit{rs}$
obtains $\Gamma,\gamma,p\vdash_g \langle \mathit{rs},s\rangle \Rightarrow t$
| $\mathit{rs}_1\ \mathit{rs}_2\ m'$ **where** $\mathit{rs} = \mathit{rs}_1\ @\ \mathit{Rule}\ m'\ \mathit{Return}\ \#\ \mathit{rs}_2$ $\mathit{matches}\ \gamma\ m'\ p$ $\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_1,s\rangle \Rightarrow \mathit{Undecided}$ *no-matching-Goto* $\gamma\ p\ \mathit{rs}_1$ $t = \mathit{Undecided}$
⟨*proof*⟩
end

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

private lemma *seq'*:
assumes $\mathit{rs} = \mathit{rs}_1\ @\ \mathit{rs}_2$ $\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_1,s\rangle \Rightarrow t$ $\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_2,t\rangle \Rightarrow t'$ **and** *no-matching-Goto* $\gamma\ p\ \mathit{rs}_1$
shows $\Gamma,\gamma,p\vdash_g \langle \mathit{rs},s\rangle \Rightarrow t'$
⟨*proof*⟩ **lemma** *seq'-cons*: $\Gamma,\gamma,p\vdash_g \langle [r],s\rangle \Rightarrow t \Longrightarrow \Gamma,\gamma,p\vdash_g \langle \mathit{rs},t\rangle \Rightarrow t' \Longrightarrow$ *no-matching-Goto* $\gamma\ p\ [r] \Longrightarrow \Gamma,\gamma,p\vdash_g \langle r\#\mathit{rs},\ s\rangle \Rightarrow t'$
⟨*proof*⟩ **lemma** *no-matching-Goto-take*: *no-matching-Goto* $\gamma\ p\ \mathit{rs} \Longrightarrow$ *no-matching-Goto* $\gamma\ p\ (\mathit{take}\ n\ \mathit{rs})$
⟨*proof*⟩ **lemma** *seq-split*:
assumes $\Gamma,\gamma,p\vdash_g \langle \mathit{rs},\ s\rangle \Rightarrow t$ $\mathit{rs} = \mathit{rs}_1 @ \mathit{rs}_2$
obtains (*no-matching-Goto*) t' **where** $\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_1,s\rangle \Rightarrow t'$ $\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_2,t'\rangle \Rightarrow t$ *no-matching-Goto* $\gamma\ p\ \mathit{rs}_1$
| (*matching-Goto*) $\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_1,s\rangle \Rightarrow t$ $\neg$ *no-matching-Goto* $\gamma\ p\ \mathit{rs}_1$
⟨*proof*⟩ **lemma** *seqE*:
assumes $\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_1 @ \mathit{rs}_2,\ s\rangle \Rightarrow t$
obtains (*no-matching-Goto*) *ti* **where** $\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_1,s\rangle \Rightarrow \mathit{ti}$ $\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_2,\mathit{ti}\rangle \Rightarrow t$ *no-matching-Goto* $\gamma\ p\ \mathit{rs}_1$
| (*matching-Goto*) $\Gamma,\gamma,p\vdash_g \langle \mathit{rs}_1,s\rangle \Rightarrow t$ $\neg$ *no-matching-Goto* $\gamma\ p\ \mathit{rs}_1$
⟨*proof*⟩ **lemma** *seqE-cons*:
assumes $\Gamma,\gamma,p\vdash_g \langle r\#\mathit{rs},\ s\rangle \Rightarrow t$
obtains (*no-matching-Goto*) *ti* **where** $\Gamma,\gamma,p\vdash_g \langle [r],s\rangle \Rightarrow \mathit{ti}$ $\Gamma,\gamma,p\vdash_g \langle \mathit{rs},\mathit{ti}\rangle \Rightarrow$

t *no-matching-Goto* γ p $[r]$
| (*matching-Goto*) $\Gamma,\gamma,p\vdash_g \langle[r],s\rangle \Rightarrow t$ $\neg$ *no-matching-Goto* γ p $[r]$
⟨*proof*⟩ **lemma** *seqE-cons-Undecided*:
assumes $\Gamma,\gamma,p\vdash_g \langle r\#rs,\ Undecided\rangle \Rightarrow t$
obtains (*no-matching-Goto*) *ti* **where** $\Gamma,\gamma,p\vdash_g \langle[r],Undecided\rangle \Rightarrow ti$ **and** $\Gamma,\gamma,p\vdash_g \langle rs,ti\rangle \Rightarrow t$ **and** *no-matching-Goto* γ p $[r]$
| (*matching-Goto*) *m chain rs′* **where** r = *Rule m* (*Goto chain*) **and** $\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ (Goto\ chain)],Undecided\rangle \Rightarrow t$ **and** *matches* γ *m p* Γ *chain* = *Some rs′*
⟨*proof*⟩ **lemma** *nomatch′*:
assumes $\bigwedge r.\ r \in set\ rs \Longrightarrow \neg\ matches\ \gamma\ (get\text{-}match\ r)\ p$
shows $\Gamma,\gamma,p\vdash_g \langle rs,\ s\rangle \Rightarrow s$
⟨*proof*⟩ **lemma** *no-free-return*: **assumes** $\Gamma,\gamma,p\vdash_g \langle[Rule\ m\ Return],\ Undecided\rangle \Rightarrow t$ **and** *matches* γ *m p* **shows** *False*
⟨*proof*⟩

17.2 Determinism

private lemma *iptables-goto-bigstep-Undecided-Undecided-deterministic*:
$\Gamma,\gamma,p\vdash_g \langle rs,\ Undecided\rangle \Rightarrow Undecided \Longrightarrow \Gamma,\gamma,p\vdash_g \langle rs,\ Undecided\rangle \Rightarrow t \Longrightarrow t = Undecided$
⟨*proof*⟩ **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*⟩ **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** $t = t'$
⟨*proof*⟩

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*⟩ **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*⟩ **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*⟩ **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*⟩ **definition** *add-match* :: *′a match-expr* ⇒ *′a rule list* ⇒ *′a rule list* **where**
add-match m rs = *map* ($\lambda r.$ *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*

⟨*proof*⟩ **lemma** *add-match-split-fst*: *add-match m* (*Rule m′ a′* # *rs*) = *Rule* (*MatchAnd m m′*) *a′* # *add-match m rs*

⟨*proof*⟩ **lemma** *matches-add-match-no-matching-Goto-simp*: *matches* γ *m p* $\Longrightarrow$ *no-matching-Goto* γ *p* (*add-match m rs*) $\Longrightarrow$ *no-matching-Goto* γ *p rs*

⟨*proof*⟩ **lemma** *matches-add-match-no-matching-Goto-simp2*: *matches* γ *m p* $\Longrightarrow$ *no-matching-Goto* γ *p rs* $\Longrightarrow$ *no-matching-Goto* γ *p* (*add-match m rs*)

⟨*proof*⟩ **lemma** *matches-add-match-MatchNot-no-matching-Goto-simp*: ¬ *matches* γ *m p* $\Longrightarrow$ *no-matching-Goto* γ *p* (*add-match m rs*)

⟨*proof*⟩ **lemma** *not-matches-add-match-simp*:
assumes ¬ *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash_g \langle$*add-match m rs*, *Undecided*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle[]$, *Undecided*$\rangle \Rightarrow t$

⟨*proof*⟩ **lemma** *matches-add-match-MatchNot-simp*:
assumes *m*: *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash_g \langle$*add-match* (*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*⟩ **lemma** *just-show-all-bigstep-semantics-equalities-with-start-Undecided*:

$\Gamma,\gamma,p\vdash_g \langle$*rs1*, *Undecided*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle$*rs2*, *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*$\rangle \Rightarrow t$

⟨*proof*⟩ **lemma** *matches-add-match-simp-helper*:
assumes *m*: *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash_g \langle$*add-match m rs*, *Undecided*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle$*rs*, *Undecided*$\rangle \Rightarrow t$ (**is** *?l* $\longleftrightarrow$ *?r*)

⟨*proof*⟩ **lemma** *matches-add-match-simp*:
matches γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash_g \langle$*add-match m rs*, *s*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle$*rs*, *s*$\rangle \Rightarrow t$

⟨*proof*⟩ **lemma** *not-matches-add-matchNot-simp*:
¬ *matches* γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash_g \langle$*add-match* (*MatchNot m*) *rs*, *s*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle$*rs*, *s*$\rangle \Rightarrow t$

⟨*proof*⟩

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 $\Gamma,\gamma,p\vdash_g \langle$(*Rule m* (*Goto chain*))#*rest*, *Undecided*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle$*add-match m rs* @ *add-match* (*MatchNot m*) *rest*, *Undecided*$\rangle \Rightarrow t$
(**is** *?l* $\longleftrightarrow$ *?r*)

⟨*proof*⟩ **theorem** *unfold-Goto*:
assumes *chain-defined*: Γ *chain* = *Some rs* **and** *no-matching-Goto-rs*: *no-matching-Goto* γ *p rs*
shows $\Gamma,\gamma,p\vdash_g \langle$(*Rule m* (*Goto chain*))#*rest*, *s*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash_g \langle$*add-match m rs* @ *add-match* (*MatchNot m*) *rest*, *s*$\rangle \Rightarrow t$

⟨*proof*⟩

A chain that will definitely come to a direct decision

qualified fun *terminal-chain* :: *'a rule list* ⇒ *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* ⟹ *no-matching-Goto* γ *p rs*
⟨*proof*⟩

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* ⟹ $\Gamma,\gamma,p\vdash_g$ ⟨*rs*, *Undecided*⟩ ⇒ *t* ⟹ ∃ *X*. *t* = *Decision X*
⟨*proof*⟩ **lemma** *replace-Goto-with-Call-in-terminal-chain-Undecided*:
assumes *chain-defined*: Γ *chain* = *Some rs* **and** *terminal-chain*: *terminal-chain rs*
shows $\Gamma,\gamma,p\vdash_g$ ⟨[*Rule m* (*Goto chain*)], *Undecided*⟩ ⇒ *t* ⟷ $\Gamma,\gamma,p\vdash_g$ ⟨[*Rule m* (*Call chain*)], *Undecided*⟩ ⇒ *t*
(**is** *?l* ⟷ *?r*)
⟨*proof*⟩ **theorem** *replace-Goto-with-Call-in-terminal-chain*:
assumes *chain-defined*: Γ *chain* = *Some rs* **and** *terminal-chain*: *terminal-chain rs*
shows $\Gamma,\gamma,p\vdash_g$ ⟨[*Rule m* (*Goto chain*)], *s*⟩ ⇒ *t* ⟷ $\Gamma,\gamma,p\vdash_g$ ⟨[*Rule m* (*Call chain*)], *s*⟩ ⇒ *t*
⟨*proof*⟩ **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 (λ*rs*. *Rule m* (*Call chain*) # *rs*) (*rewrite-Goto-chain-safe* Γ *rs*)
)
) |
rewrite-Goto-chain-safe Γ (*r*#*rs*) = *map-option* (λ*rs*. *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* (λ*rst*. (*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*: ($\bigwedge$*s*. Γ,γ,*p*$\vdash_g$ ⟨*rs1*, *s*⟩ ⇒ *t* = Γ,γ,*p*$\vdash_g$ ⟨*rs2*, *s*⟩ ⇒ *t*) ⟹
Γ,γ,*p*$\vdash_g$ ⟨*r*#*rs1*, *s*⟩ ⇒ *t* = Γ,γ,*p*$\vdash_g$ ⟨*r*#*rs2*, *s*⟩ ⇒ *t*
⟨*proof*⟩ **lemma** *terminal-chain-decision*:
terminal-chain rs ⟹ Γ,γ,*p*$\vdash_g$ ⟨*rs*, *Undecided*⟩ ⇒ *t* ⟹ ∃ *X*. *t* = *Decision X*
⟨*proof*⟩ **lemma** *terminal-chain-Goto-decision*: Γ *chain* = *Some rs* ⟹ *terminal-chain rs* ⟹ *matches* γ *m p* ⟹
Γ,γ,*p*$\vdash_g$ ⟨[*Rule m* (*Goto chain*)], *s*⟩ ⇒ *t* ⟹ ∃ *X*. *t* = *Decision X*
⟨*proof*⟩ **theorem** *rewrite-Goto-chain-safe*:
rewrite-Goto-chain-safe Γ *rs* = *Some rs′* ⟹ Γ,γ,*p*$\vdash_g$ ⟨*rs′*, *s*⟩ ⇒ *t* ⟷ Γ,γ,*p*$\vdash_g$ ⟨*rs*, *s*⟩ ⇒ *t*
⟨*proof*⟩

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*$\vdash_g$⟨[*Rule MatchAny* (*Call* ″*FORWARD*″)], *Undecided*⟩ ⇒ (*Decision FinalAllow*)
⟨*proof*⟩

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*
⟨*proof*⟩

representing *False*

definition *dnf-False* :: *'a dnf* **where**
dnf-False ≡ []
lemma *dnf-False*: ¬ *dnf-to-bool f dnf-False*
⟨*proof*⟩

lemma *cnf-to-bool-append*: *cnf-to-bool* γ (*a1* @ *a2*) ⟷ *cnf-to-bool* γ *a1* ∧ *cnf-to-bool* γ *a2*
⟨*proof*⟩
lemma *dnf-to-bool-append*: *dnf-to-bool* γ (*a1* @ *a2*) ⟷ *dnf-to-bool* γ *a1* ∨ *dnf-to-bool* γ *a2*
⟨*proof*⟩

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*⟩
lemma *dnf-to-bool-set*: *dnf-to-bool* γ *dnf* ⟷ (∃ *d* ∈ *set dnf*. *cnf-to-bool* γ *d*)
⟨*proof*⟩

lemma *dnf-to-bool-seteq*: *set ‘ set d1* = *set ‘ set d2* ⟹ *dnf-to-bool γ d1* ⟷ *dnf-to-bool γ d2*
⟨*proof*⟩

lemma *dnf-and-correct*: *dnf-to-bool γ* (*dnf-and d1 d2*) ⟷ *dnf-to-bool γ d1* ∧ *dnf-to-bool γ d2*
⟨*proof*⟩

lemma *dnf-and-symmetric*: *dnf-to-bool γ* (*dnf-and d1 d2*) ⟷ *dnf-to-bool γ* (*dnf-and d2 d1*)
⟨*proof*⟩

18.0.1 inverting a DNF

Example

lemma (¬ ((*a1* ∧ *a2*) ∨ *b* ∨ *c*)) = ((¬*a1* ∧ ¬ *b* ∧ ¬ *c*) ∨ (¬*a2* ∧ ¬ *b* ∧ ¬ *c*)) ⟨*proof*⟩
lemma (¬ ((*a1* ∧ *a2*) ∨ (*b1* ∧ *b2*) ∨ *c*)) = ((¬*a1* ∧ ¬ *b1* ∧ ¬ *c*) ∨ (¬*a2* ∧ ¬ *b1* ∧ ¬ *c*) ∨ (¬*a1* ∧ ¬ *b2* ∧ ¬ *c*) ∨ (¬*a2* ∧ ¬ *b2* ∧ ¬ *c*)) ⟨*proof*⟩

fun *listprepend* :: *'a list* ⇒ *'a list list* ⇒ *'a list list* **where**
listprepend [] *ns* = [] |
listprepend (*a#as*) *ns* = (*map* (*λxs. a#xs*) *ns*) @ (*listprepend as ns*)

lemma *listprepend* [*a*,*b*] [*as*, *bs*] = [*a#as*, *a#bs*, *b#as*, *b#bs*] ⟨*proof*⟩

lemma *map-a-and*: *dnf-to-bool γ* (*map* ((#) *a*) *ds*) ⟷ *dnf-to-bool γ* [[*a*]] ∧ *dnf-to-bool γ ds*
⟨*proof*⟩

this is how *listprepend* works:

lemma ¬ *dnf-to-bool γ* (*listprepend* [] *ds*) ⟨*proof*⟩
lemma *dnf-to-bool γ* (*listprepend* [*a*] *ds*) ⟷ *dnf-to-bool γ* [[*a*]] ∧ *dnf-to-bool γ ds* ⟨*proof*⟩
lemma *dnf-to-bool γ* (*listprepend* [*a*, *b*] *ds*) ⟷ (*dnf-to-bool γ* [[*a*]] ∧ *dnf-to-bool γ ds*) ∨ (*dnf-to-bool γ* [[*b*]] ∧ *dnf-to-bool γ ds*)
⟨*proof*⟩

We use ∃ to model the big ∨ operation

lemma *listprepend-correct*: *dnf-to-bool γ* (*listprepend as ds*) ⟷ (∃ *a*∈ *set as*. *dnf-to-bool γ* [[*a*]] ∧ *dnf-to-bool γ ds*)
⟨*proof*⟩
lemma *listprepend-correct'*: *dnf-to-bool γ* (*listprepend as ds*) ⟷ (*dnf-to-bool γ* (*map* (*λa.* [*a*]) *as*) ∧ *dnf-to-bool γ ds*)
⟨*proof*⟩

lemma *cnf-invert-singelton*: *cnf-to-bool γ* [*invert a*] ⟷ ¬ *cnf-to-bool γ* [*a*] ⟨*proof*⟩

lemma *cnf-singleton-false*: $(\exists\, a' \in set\ as.\ \neg\ cnf\text{-}to\text{-}bool\ \gamma\ [a']) \longleftrightarrow \neg\ cnf\text{-}to\text{-}bool\ \gamma\ as$
⟨*proof*⟩

fun *dnf-not* :: *'a dnf* ⇒ *'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*) ⟷ ¬ *dnf-to-bool* γ *d*
⟨*proof*⟩

18.0.2 Optimizing

definition *optimize-dfn* :: *'a dnf* ⇒ *'a dnf* **where**
optimize-dfn dnf = *map remdups* (*remdups dnf*)

lemma *dnf-to-bool f* (*optimize-dfn dnf*) = *dnf-to-bool f dnf*
⟨*proof*⟩

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* (β, α) *m a p* ≠ *Semantics.matches* (λ *atm p. case* β *atm p of TernaryTrue* ⇒ *True* | *TernaryFalse* ⇒ *False* | *TernaryUnknown* ⇒ α *a p*) *m p*
⟨*proof*⟩

the *the* in the next definition is always defined

lemma $\forall\, m \in$ {*m. approx m p* ≠ *TernaryUnknown*}. *ternary-to-bool* (*approx m p*) ≠ *None*
⟨*proof*⟩

The Boolean and the ternary matcher agree (where the ternary matcher is defined)

definition *matcher-agree-on-exact-matches* :: (*'a*, *'p*) *matcher* ⇒ (*'a* ⇒ *'p* ⇒ *ternaryvalue*) ⇒ *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*)
⟨*proof*⟩
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*)
⟨*proof*⟩

lemma *eval-ternary-Not-TrueD*: *eval-ternary-Not m* $=$ *TernaryTrue* $\Longrightarrow$ *m* $=$ *TernaryFalse*
⟨*proof*⟩

lemma *matches-comply-exact*: *ternary-ternary-eval* (*map-match-tac* β *p m*) $\neq$ *TernaryUnknown* $\Longrightarrow$
matcher-agree-on-exact-matches γ β $\Longrightarrow$
Semantics.matches γ *m p* $=$ *Matching-Ternary.matches* (β, α) *m a p*
⟨*proof*⟩

lemma *matcher-agree-on-exact-matches-gammaE*:
matcher-agree-on-exact-matches γ β $\Longrightarrow$ β *X p* $=$ *TernaryTrue* $\Longrightarrow$ γ *X p*
⟨*proof*⟩

lemma *in-doubt-allow-allows-Accept*: *a* $=$ *Accept* $\Longrightarrow$ *matcher-agree-on-exact-matches* γ β $\Longrightarrow$
Semantics.matches γ *m p* $\Longrightarrow$ *Matching-Ternary.matches* (β, *in-doubt-allow*) *m a p*
⟨*proof*⟩

lemma *not-exact-match-in-doubt-allow-approx-match*: *matcher-agree-on-exact-matches* γ β $\Longrightarrow$ *a* $=$ *Accept* $\vee$ *a* $=$ *Reject* $\vee$ *a* $=$ *Drop* $\Longrightarrow$
$\neg$ *Semantics.matches* γ *m p* $\Longrightarrow$
(*a* $=$ *Accept* $\wedge$ *Matching-Ternary.matches* (β, *in-doubt-allow*) *m a p*) $\vee$ $\neg$ *Matching-Ternary.matches* (β, *in-doubt-allow*) *m a p*
⟨*proof*⟩

lemma *in-doubt-deny-denies-DropReject*: *a* $=$ *Drop* $\vee$ *a* $=$ *Reject* $\Longrightarrow$ *matcher-agree-on-exact-matches*

$\gamma\ \beta \Longrightarrow$
Semantics.matches γ *m p* $\Longrightarrow$ *Matching-Ternary.matches* (β, *in-doubt-deny*) *m a p*
⟨*proof*⟩

lemma *not-exact-match-in-doubt-deny-approx-match*: *matcher-agree-on-exact-matches* $\gamma\ \beta \Longrightarrow$ *a* = *Accept* ∨ *a* = *Reject* ∨ *a* = *Drop* $\Longrightarrow$
¬ *Semantics.matches* γ *m p* $\Longrightarrow$
((*a* = *Drop* ∨ *a* = *Reject*) ∧ *Matching-Ternary.matches* (β, *in-doubt-deny*) *m a p*) ∨ ¬ *Matching-Ternary.matches* (β, *in-doubt-deny*) *m a p*
⟨*proof*⟩

The ternary primitive matcher can return exactly the result of the Boolean primitive matcher

definition β_{magic} :: (*'a*, *'p*) *matcher* ⇒ (*'a* ⇒ *'p* ⇒ *ternaryvalue*) **where**
$\beta_{magic}\ \gamma$ ≡ (λ *a p*. *if* γ *a p then TernaryTrue else TernaryFalse*)

lemma *matcher-agree-on-exact-matches* γ ($\beta_{magic}\ \gamma$)
⟨*proof*⟩

lemma β_{magic}-*not-Unknown*: *ternary-ternary-eval* (*map-match-tac* ($\beta_{magic}\ \gamma$) *p m*) ≠ *TernaryUnknown*
⟨*proof*⟩

lemma β_{magic}-*matching*: *Matching-Ternary.matches* (($\beta_{magic}\ \gamma$), α) *m a p* ⟷ *Semantics.matches* γ *m p*
⟨*proof*⟩

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*
⟨*proof*⟩

utility lemmas

context
begin

private lemma *fixedaction-Log*: *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m Log*) *ms*) *Undecided* = *Undecided*
⟨*proof*⟩ **lemma** *fixedaction-Empty*:*approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m Empty*) *ms*) *Undecided* = *Undecided*
⟨*proof*⟩ **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*
⟨*proof*⟩ **lemma** *helperX1-Empty*: *matches* γ *m'* *Empty p* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*map* ((λ*m. Rule m Empty*) $\circ$ *MatchAnd m'*) *m2'* @ *rs2*) *Undecided* =
approximating-bigstep-fun γ *p rs2 Undecided*
⟨*proof*⟩ **lemma** *helperX3*: *matches* γ *m'* *a p* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*map* ((λ*m. Rule m a*) $\circ$ *MatchAnd m'*) *m2'* @ *rs2*) *Undecided* =
approximating-bigstep-fun γ *p* (*map* (λ*m. Rule m a*) *m2'* @ *rs2*) *Undecided*
⟨*proof*⟩

lemmas *fixed-action-simps* = *fixedaction-Log fixedaction-Empty helperX1-Log helperX1-Empty helperX3*
end

lemma *fixedaction-swap*:
approximating-bigstep-fun γ *p* (*map* (λ*m. Rule m a*) (*m1*@*m2*)) *s* = *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) (*m2*@*m1*)) *s*
⟨*proof*⟩

corollary *fixedaction-reorder*: *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) (*m1* @ *m2* @ *m3*)) *s* = *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) (*m2* @ *m1* @ *m3*)) *s*
⟨*proof*⟩

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* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*map* (λ*m. Rule m a*) *m1*) *s* =
approximating-bigstep-fun γ *p* (*map* (λ*m. Rule m a*) *m2*) *s*
⟨*proof*⟩

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*: $match\text{-}list\ \gamma\ ms\ a\ p \longleftrightarrow (\exists\, m \in set\ ms.\ matches\ \gamma\ m\ a\ p)$
⟨*proof*⟩

lemma *match-list-True*: $match\text{-}list\ \gamma\ ms\ a\ p \Longrightarrow approximating\text{-}bigstep\text{-}fun\ \gamma\ p\ (map\ (\lambda m.\ Rule\ m\ a)\ ms)\ Undecided = (case\ a\ of\ Accept \Rightarrow Decision\ FinalAllow$
$\mid Drop \Rightarrow Decision\ FinalDeny$
$\mid Reject \Rightarrow Decision\ FinalDeny$
$\mid Log \Rightarrow Undecided$
$\mid Empty \Rightarrow Undecided$
— unhandled cases
)
⟨*proof*⟩
lemma *match-list-False*: $\neg\ match\text{-}list\ \gamma\ ms\ a\ p \Longrightarrow approximating\text{-}bigstep\text{-}fun\ \gamma\ p\ (map\ (\lambda m.\ Rule\ m\ a)\ ms)\ Undecided = Undecided$
⟨*proof*⟩

The key idea behind *match-list*: Reducing semantics to match list

lemma *match-list-semantics*: $match\text{-}list\ \gamma\ ms1\ a\ p \longleftrightarrow match\text{-}list\ \gamma\ ms2\ a\ p \Longrightarrow approximating\text{-}bigstep\text{-}fun\ \gamma\ p\ (map\ (\lambda m.\ Rule\ m\ a)\ ms1)\ s = approximating\text{-}bigstep\text{-}fun\ \gamma\ p\ (map\ (\lambda m.\ Rule\ m\ a)\ ms2)\ s$
⟨*proof*⟩

We can exploit de-morgan to get a disjunction in the match expression!

fun *match-list-to-match-expr* :: *'a match-expr list* ⇒ *'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*: $match\text{-}list\ \gamma\ ms\ a\ p \longleftrightarrow matches\ \gamma\ (match\text{-}list\text{-}to\text{-}match\text{-}expr\ ms)\ a\ p$
⟨*proof*⟩

lemma *match-list-singleton*: $match\text{-}list\ \gamma\ [m]\ a\ p \longleftrightarrow matches\ \gamma\ m\ a\ p$ ⟨*proof*⟩

lemma *match-list-append*: $match\text{-}list\ \gamma\ (m1 @ m2)\ a\ p \longleftrightarrow (\neg\ match\text{-}list\ \gamma\ m1\ a\ p \longrightarrow match\text{-}list\ \gamma\ m2\ a\ p)$
⟨*proof*⟩

lemma *match-list-helper1*: $\neg\ matches\ \gamma\ m2\ a\ p \Longrightarrow match\text{-}list\ \gamma\ (map\ (\lambda x.\ MatchAnd\ x\ m2)\ m1')\ a\ p \Longrightarrow False$
⟨*proof*⟩
lemma *match-list-helper2*: $\neg\ matches\ \gamma\ m\ a\ p \Longrightarrow \neg\ match\text{-}list\ \gamma\ (map\ (MatchAnd\ m)\ m2')\ a\ p$

⟨*proof*⟩

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*

⟨*proof*⟩

lemma *match-list-helper4*: $\neg$ *match-list* γ *m2'* *a p* $\Longrightarrow$ $\neg$ *match-list* γ (*map* (*MatchAnd aa*) *m2'*) *a p*

⟨*proof*⟩

lemma *match-list-helper5*: $\neg$ *match-list* γ *m2'* *a p* $\Longrightarrow$ $\neg$ *match-list* γ (*concat* (*map* (λx. *map* (*MatchAnd x*) *m2'*) *m1'*)) *a p*

⟨*proof*⟩

lemma *match-list-helper6*: $\neg$ *match-list* γ *m1'* *a p* $\Longrightarrow$ $\neg$ *match-list* γ (*concat* (*map* (λx. *map* (*MatchAnd x*) *m2'*) *m1'*)) *a p*

⟨*proof*⟩

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* (λx. *MatchAnd x m2*) *m1'*) *a p*

⟨*proof*⟩

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*

⟨*proof*⟩

lemma *match-list-concat*: *match-list* γ (*concat lss*) *a p* $\longleftrightarrow$ ($\exists$ *ls* $\in$ *set lss*. *match-list* γ *ls a p*)

⟨*proof*⟩

lemma *fixedaction-wf-ruleset*: *wf-ruleset* γ *p* (*map* (λm. *Rule m a*) *ms*) $\longleftrightarrow$
$\neg$ *match-list* γ *ms a p* $\vee$ $\neg$ ($\exists$ *chain*. *a* = *Call chain*) $\wedge$ *a* $\neq$ *Return* $\wedge$ $\neg$ ($\exists$ *chain*. *a* = *Goto chain*) $\wedge$ *a* $\neq$ *Unknown*

⟨*proof*⟩

lemma *wf-ruleset-singleton*: *wf-ruleset* γ *p* [*Rule m a*] $\longleftrightarrow$ $\neg$ *matches* γ *m a p* $\vee$ $\neg$ ($\exists$ *chain*. *a* = *Call chain*) $\wedge$ *a* $\neq$ *Return* $\wedge$ $\neg$ ($\exists$ *chain*. *a* = *Goto chain*) $\wedge$ *a* $\neq$ *Unknown*

⟨*proof*⟩

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 ∨ of the list elements.

fun *normalize-match* :: *'a match-expr* ⇒ *'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*: ∀ *m'* ∈ *set* (*normalize-match m*). ¬ *matcheq-matchNone m'*
⟨*proof*⟩

lemma *normalize-match-empty-iff-matcheq-matchNone*: *normalize-match m* = [] ⟷ *matcheq-matchNone m*
⟨*proof*⟩

lemma *match-list-normalize-match*: *match-list* γ [*m*] *a p* ⟷ *match-list* γ (*normalize-match m*) *a p*
⟨*proof*⟩

thm *match-list-normalize-match*[*simplified match-list-singleton*]

theorem *normalize-match-correct*: *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) (*normalize-match m*)) *s* = *approximating-bigstep-fun* γ *p* [*Rule m a*] *s*
⟨*proof*⟩

lemma *normalize-match-empty*: *normalize-match m* = [] ⟹ ¬ *matches* γ *m a p*
⟨*proof*⟩

lemma *matches-to-match-list-normalize*: *matches* γ *m a p* = *match-list* γ (*normalize-match m*) *a p*
⟨*proof*⟩

lemma *wf-ruleset-normalize-match*: *wf-ruleset* γ *p* [(*Rule m a*)] $\Longrightarrow$ *wf-ruleset* γ *p* (*map* (λ*m. Rule m a*) (*normalize-match m*))
⟨*proof*⟩

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*⟩

lemma *good-ruleset-normalize-match*: *good-ruleset* [(*Rule m a*)] $\Longrightarrow$ *good-ruleset* (*map* (λ*m. Rule m a*) (*normalize-match m*))
⟨*proof*⟩

22 Normalizing rules instead of only match expressions

fun *normalize-rules* :: (*'a match-expr* $\Rightarrow$ *'a match-expr list*) $\Rightarrow$ *'a rule list* $\Rightarrow$ *'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*) ⟨*proof*⟩
lemma *normalize-rules-fst*: (*normalize-rules f* (*r* # *rs*)) = (*normalize-rules f* [*r*]) @ (*normalize-rules f rs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *good-ruleset-normalize-rules*: *good-ruleset rs* $\Longrightarrow$ *good-ruleset* (*normalize-rules f rs*)
⟨*proof*⟩

lemma *simple-ruleset-normalize-rules*: *simple-ruleset rs* $\Longrightarrow$ *simple-ruleset* (*normalize-rules f rs*)
⟨*proof*⟩

lemma *normalize-rules-match-list-semantics-3*:
assumes $\forall$ *m a. P m* $\longrightarrow$ *match-list* γ (*f m*) *a p* = *matches* γ *m a p*
and *simple-ruleset rs*
and *P*: $\forall$ *r* $\in$ *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*⟩

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*
⟨*proof*⟩

lemma *in-normalized-matches*: *ls* $\in$ *set* (*normalize-match m*) $\wedge$ *matches* γ *ls a p* $\Longrightarrow$ *matches* γ *m a p*
⟨*proof*⟩

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*⟩

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** $\forall$ *r* $\in$ *set rs*. *P* (*get-match r*)
and $\forall$ *m*. *P m* $\longrightarrow$ ($\forall$ *m'* $\in$ *set* (*f m*). *P m'*)
shows $\forall$ *r* $\in$ *set* (*normalize-rules f rs*). *P* (*get-match r*)
⟨*proof*⟩

fun *normalize-rules-dnf* :: *'a rule list* $\Rightarrow$ *'a rule list* **where**
normalize-rules-dnf [] = [] |
normalize-rules-dnf ((*Rule m a*)#*rs*) = (*map* (λ*m*. *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*⟩

lemma *normalize-rules-dnf-def2*: *normalize-rules-dnf* = *normalize-rules normalize-match*
⟨*proof*⟩

lemma *wf-ruleset-normalize-rules-dnf*: *wf-ruleset* γ *p rs* $\Longrightarrow$ *wf-ruleset* γ *p* (*normalize-rules-dnf rs*)
⟨*proof*⟩

lemma *good-ruleset-normalize-rules-dnf*: *good-ruleset rs* $\Longrightarrow$ *good-ruleset* (*normalize-rules-dnf rs*)
⟨*proof*⟩

lemma *simple-ruleset-normalize-rules-dnf*: *simple-ruleset rs* $\Longrightarrow$ *simple-ruleset* (*normalize-rules-dnf rs*)
⟨*proof*⟩

lemma *simple-ruleset rs* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*normalize-rules-dnf rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
⟨*proof*⟩

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*⟩

fun *normalized-nnf-match* :: *'a match-expr* $\Rightarrow$ *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*) $\wedge$ (*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* $\Longrightarrow$ *normalize-match m* = [*m*]
⟨*proof*⟩

lemma *normalized-nnf-match-normalize-match*: $\forall$ *m'* $\in$ *set* (*normalize-match m*). *normalized-nnf-match m'*
⟨*proof*⟩

lemma *normalized-nnf-match-MatchNot-D*: *normalized-nnf-match* (*MatchNot m*) $\Longrightarrow$ *normalized-nnf-match m*
⟨*proof*⟩

Example

lemma *normalize-match* (*MatchNot* (*MatchAnd* (*Match ip-src*) (*Match tcp*))) = [*MatchNot* (*Match ip-src*), *MatchNot* (*Match tcp*)] ⟨*proof*⟩

22.1 Functions which preserve *normalized-nnf-match*

lemma *optimize-matches-option-normalized-nnf-match*: $(\bigwedge r.\ r \in set\ rs \Longrightarrow normalized\text{-}nnf\text{-}match\ (get\text{-}match\ r)) \Longrightarrow$
$(\bigwedge m\ m'.\ normalized\text{-}nnf\text{-}match\ m \Longrightarrow f\ m = Some\ m' \Longrightarrow normalized\text{-}nnf\text{-}match\ m') \Longrightarrow$
$\forall\ r \in set\ (optimize\text{-}matches\text{-}option\ f\ rs).\ normalized\text{-}nnf\text{-}match\ (get\text{-}match\ r)$
⟨*proof*⟩

lemma *optimize-matches-normalized-nnf-match*: $[\![\forall\ r \in set\ rs.\ normalized\text{-}nnf\text{-}match\ (get\text{-}match\ r); \forall\ m.\ normalized\text{-}nnf\text{-}match\ m \longrightarrow normalized\text{-}nnf\text{-}match\ (f\ m)\ ]\!] \Longrightarrow$
$\forall\ r \in set\ (optimize\text{-}matches\ f\ rs).\ normalized\text{-}nnf\text{-}match\ (get\text{-}match\ r)$
⟨*proof*⟩

lemma *normalize-rules-dnf-normalized-nnf-match*: $\forall\ x \in set\ (normalize\text{-}rules\text{-}dnf\ rs).\ normalized\text{-}nnf\text{-}match\ (get\text{-}match\ x)$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *alist-and-append*: $matches\ \gamma\ (alist\text{-}and\ (l1\ @\ l2))\ a\ p \longleftrightarrow matches\ \gamma\ (MatchAnd\ (alist\text{-}and\ l1)\ (alist\text{-}and\ l2))\ a\ p$
⟨*proof*⟩

This version of *alist-and* avoids the trailing *MatchAny*. Only intended for code.

fun *alist-and'* :: *'a negation-type list* ⇒ *'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*)
⟨*proof*⟩

lemma *normalized-nnf-match-alist-and'*: *normalized-nnf-match* (*alist-and' as*)
⟨*proof*⟩

lemma *matches-alist-and-alist-and'*:
matches γ (*alist-and' ls*) *a p* ⟷ *matches* γ (*alist-and ls*) *a p*
⟨*proof*⟩

lemma *alist-and'-append*: *matches* γ (*alist-and'* (*l1* @ *l2*)) *a p* ⟷ *matches* γ (*MatchAnd* (*alist-and' l1*) (*alist-and' l2*)) *a p*
⟨*proof*⟩

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*)
⟷
matches γ (*alist-and* (*NegPos-map C spts*)) *a p*
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

lemma *alist-and-negation-type-to-match-expr*: *alist-and* (*n*#*es*) = *MatchAnd* (*negation-type-to-match-expr n*) (*alist-and es*)
⟨*proof*⟩

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* $\Longrightarrow$ *matches* γ (*alist-and* (*to-negation-type-nnf m*)) *a p* = *matches* γ *m a p*
⟨*proof*⟩

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* ⟷ *matches* γ (*alist-and l*) *a p*
⟨*proof*⟩

lemma *nt-match-list-simp*: *nt-match-list* γ *a p ms* ⟷
(∀ *m* ∈ *set* (*getPos ms*). *matches* γ (*Match m*) *a p*) ∧ (∀ *m* ∈ *set* (*getNeg ms*). *matches* γ (*MatchNot* (*Match m*)) *a p*)
⟨*proof*⟩

lemma *matches-alist-and*: *matches* γ (*alist-and l*) *a p* ⟷ (∀ *m* ∈ *set* (*getPos l*). *matches* γ (*Match m*) *a p*) ∧ (∀ *m* ∈ *set* (*getNeg l*). *matches* γ (*MatchNot* (*Match m*)) *a p*)
⟨*proof*⟩

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-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-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)))* ⟨*proof*⟩
lemma *has-disc-negated* (λx::*nat*. $x = 0$) *False (MatchAnd (Match 0) (MatchNot (Match 0)))* ⟨*proof*⟩
lemma *has-disc-negated* (λx::*nat*. $x = 0$) *True (MatchAnd (Match 0) (MatchNot (Match 1)))* ⟨*proof*⟩
lemma $\neg$ *has-disc-negated* (λx::*nat*. $x = 0$) *True (MatchAnd (Match 1) (MatchNot (Match 0)))* ⟨*proof*⟩
lemma *has-disc-negated* (λx::*nat*. $x = 0$) *True (MatchAnd (Match 0) (MatchNot (Match 0)))* ⟨*proof*⟩
lemma *has-disc-negated-MatchNot*:
has-disc-negated disc True (MatchNot m) $\longleftrightarrow$ *has-disc-negated disc False m*
has-disc-negated disc True m $\longleftrightarrow$ *has-disc-negated disc False (MatchNot m)*
⟨*proof*⟩

lemma *has-disc-negated-has-disc*: *has-disc-negated disc neg m* $\Longrightarrow$ *has-disc disc m*
⟨*proof*⟩

lemma *has-disc-negated-positiv-has-disc*: *has-disc-negated disc neg m* $\vee$ *has-disc-negated disc* ($\neg$ *neg*) *m* $\longleftrightarrow$ *has-disc disc m*
⟨*proof*⟩

lemma *has-disc-negated-disj-split*:
has-disc-negated (λa. *P a* $\vee$ *Q a*) *neg m* $\longleftrightarrow$ *has-disc-negated P neg m* $\vee$ *has-disc-negated Q neg m*
⟨*proof*⟩

lemma *has-disc-alist-and*: *has-disc disc (alist-and as)* $\longleftrightarrow$ ($\exists$ $a \in$ *set as. has-disc disc (negation-type-to-match-expr a)*)

⟨*proof*⟩
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*⟩

lemma *has-disc-alist-and'*: *has-disc disc* (*alist-and' as*) ⟷ (∃ *a* ∈ *set as*. *has-disc disc* (*negation-type-to-match-expr a*))
⟨*proof*⟩
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*⟩

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*)
⟨*proof*⟩
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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *matches* ((λ*x* -. *bool-to-ternary* (*disc x*)), (λ- -. *False*)) (*Match x*) *a p* ⟷ *has-disc disc* (*Match x*)
⟨*proof*⟩

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 $\Longrightarrow$ *normalized-nnf-match* (*opt-MatchAny-match-expr m*)
⟨*proof*⟩

lemma *normalized-n-primitive-opt-MatchAny-match-expr*:
normalized-n-primitive disc-sel f m $\Longrightarrow$ *normalized-n-primitive disc-sel f* (*opt-MatchAny-match-expr m*)
⟨*proof*⟩

lemma *normalized-n-primitive-imp-not-disc-negated*:
wf-disc-sel (*disc*,*sel*) *C* $\Longrightarrow$ *normalized-n-primitive* (*disc*,*sel*) *f m* $\Longrightarrow$ ¬ *has-disc-negated disc False m*
⟨*proof*⟩

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*⟩

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*⟩

lemma *not-has-disc-NegPos-map*: ∀ *a*. ¬ *disc* (*C a*) $\Longrightarrow$ ∀ *a*∈*set* (*NegPos-map C ls*).
¬ *has-disc disc* (*negation-type-to-match-expr a*)
⟨*proof*⟩

lemma *not-has-disc-negated-NegPos-map*: ∀ *a*. ¬ *disc* (*C a*) $\Longrightarrow$ ∀ *a*∈*set* (*NegPos-map C ls*).
¬ *has-disc-negated disc False* (*negation-type-to-match-expr a*)
⟨*proof*⟩

lemma *normalized-n-primitive-impossible-map*: ∀ *a*. ¬ *disc* (*C a*) $\Longrightarrow$
∀ *m*∈*set* (*map* (*Match* ∘ (*C* ∘ *x*)) *ls*).
normalized-n-primitive (*disc*, *sel*) *f m*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *normalized-n-primitive-if-no-primitive*: *normalized-nnf-match m* $\Longrightarrow$ $\neg$ *has-disc disc m* $\Longrightarrow$
normalized-n-primitive (*disc*, *sel*) *f m*
⟨*proof*⟩

lemma *normalized-n-primitive-false-eq-notdisc*: *normalized-nnf-match m* $\Longrightarrow$
normalized-n-primitive (*disc*, *sel*) (λ-. *False*) *m* $\longleftrightarrow$ $\neg$ *has-disc disc m*
⟨*proof*⟩

lemma *normalized-n-primitive-MatchAnd-combine-map*: *normalized-n-primitive disc-sel f rst* $\Longrightarrow$
$\forall$ *m'* $\in$ (λ*spt. Match* (*C spt*)) ‘ *set pts. normalized-n-primitive disc-sel f m'* $\Longrightarrow$
m' $\in$ (λ*spt. MatchAnd* (*Match* (*C spt*)) *rst*) ‘ *set pts* $\Longrightarrow$ *normalized-n-primitive disc-sel f m'*
⟨*proof*⟩

24.2 Primitive Extractor

The following function takes a tuple of functions ((*'a* $\Rightarrow$ *bool*) $\times$ (*'a* $\Rightarrow$ *'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* $\Rightarrow$ *bool*) $\times$ (*'a* $\Rightarrow$ *'b*)) $\Rightarrow$ *'a match-expr* $\Rightarrow$ (*'b negation-type list* $\times$ *'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* $\Rightarrow$ $'a$ *match-expr* $\Rightarrow$ $'a$ *match-expr*
shows *fst* (*case primitive-extractor* (*disc*, *sel*) *m1 of* (*a1'*, *ms1'*) $\Rightarrow$ *case primitive-extractor* (*disc*, *sel*) *m2 of* (*a2'*, *ms2'*) $\Rightarrow$ (*a1'* @ *a2'*, *m' ms1' ms2'*)) =
fst (*primitive-extractor* (*disc*, *sel*) *m1*) @ *fst* (*primitive-extractor* (*disc*, *sel*) *m2*)
⟨*proof*⟩

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* $\wedge$ *matches* γ *ms a p* $\longleftrightarrow$ *matches* γ *m a p*
and *normalized-nnf-match ms*
and $\neg$ *has-disc disc ms*
and $\forall$ *disc2*. $\neg$ *has-disc disc2 m* $\longrightarrow$ $\neg$ *has-disc disc2 ms*
and $\forall$ *disc2 sel2*. *normalized-n-primitive* (*disc2*, *sel2*) *P m* $\longrightarrow$ *normalized-n-primitive* (*disc2*, *sel2*) *P ms*
and $\forall$ *disc2*. $\neg$ *has-disc-negated disc2 neg m* $\longrightarrow$ $\neg$ *has-disc-negated disc2 neg ms*
and $\neg$ *has-disc disc m* $\longleftrightarrow$ *as* = [] $\wedge$ *ms* = *m*
and $\neg$ *has-disc-negated disc False m* $\longleftrightarrow$ *getNeg as* = []
and *has-disc disc m* $\Longrightarrow$ *as* $\neq$ []
⟨*proof*⟩

lemma *has-disc-negated-primitive-extractor*:
assumes *normalized-nnf-match m*
shows *has-disc-negated disc False m* $\longleftrightarrow$ ($\exists$ *a*. *Neg a* $\in$ *set* (*fst* (*primitive-extractor* (*disc*, *sel*) *m*)))
⟨*proof*⟩

lemma *primitive-extractor-reassemble-preserves*:
wf-disc-sel (*disc*, *sel*) *C* $\Longrightarrow$
normalized-nnf-match m $\Longrightarrow$
P m $\Longrightarrow$
P MatchAny $\Longrightarrow$
primitive-extractor (*disc*, *sel*) *m* = (*as*, *ms*) $\Longrightarrow$ — turn eqality around to simplify proof
($\bigwedge$*m1 m2*. *P* (*MatchAnd m1 m2*) $\longleftrightarrow$ *P m1* $\wedge$ *P m2*) $\Longrightarrow$
($\bigwedge$*ls1 ls2*. *P* (*alist-and'* (*ls1* @ *ls2*)) $\longleftrightarrow$ *P* (*alist-and' ls1*) $\wedge$ *P* (*alist-and' ls2*)) $\Longrightarrow$
P (*alist-and'* (*NegPos-map C as*))
⟨*proof*⟩

lemma *primitive-extractor-reassemble-not-has-disc*:
wf-disc-sel (*disc*, *sel*) *C* $\Longrightarrow$
normalized-nnf-match m $\Longrightarrow$ $\neg$ *has-disc disc' m* $\Longrightarrow$

$primitive\text{-}extractor\ (disc,\ sel)\ m = (as,\ ms) \Longrightarrow$
$\neg\ has\text{-}disc\ disc'\ (alist\text{-}and'\ (NegPos\text{-}map\ C\ as))$
⟨*proof*⟩

lemma *primitive-extractor-reassemble-not-has-disc-negated*:
$wf\text{-}disc\text{-}sel\ (disc,\ sel)\ C \Longrightarrow$
$normalized\text{-}nnf\text{-}match\ m \Longrightarrow \neg\ has\text{-}disc\text{-}negated\ disc'\ neg\ m \Longrightarrow$
$primitive\text{-}extractor\ (disc,\ sel)\ m = (as,\ ms) \Longrightarrow$
$\neg\ has\text{-}disc\text{-}negated\ disc'\ neg\ (alist\text{-}and'\ (NegPos\text{-}map\ C\ as))$
⟨*proof*⟩

lemma *primitive-extractor-reassemble-normalized-n-primitive*:
$wf\text{-}disc\text{-}sel\ (disc,\ sel)\ C \Longrightarrow$
$normalized\text{-}nnf\text{-}match\ m \Longrightarrow normalized\text{-}n\text{-}primitive\ (disc1,\ sel1)\ f\ m \Longrightarrow$
$primitive\text{-}extractor\ (disc,\ sel)\ m = (as,\ ms) \Longrightarrow$
$normalized\text{-}n\text{-}primitive\ (disc1,\ sel1)\ f\ (alist\text{-}and'\ (NegPos\text{-}map\ C\ as))$
⟨*proof*⟩

lemma *primitive-extractor-matchesE*: $wf\text{-}disc\text{-}sel\ (disc,sel)\ C \Longrightarrow normalized\text{-}nnf\text{-}match\ m \Longrightarrow primitive\text{-}extractor\ (disc,\ sel)\ m = (as,\ ms)$
$\Longrightarrow$
$(normalized\text{-}nnf\text{-}match\ ms \Longrightarrow \neg\ has\text{-}disc\ disc\ ms \Longrightarrow (\forall\ disc2.\ \neg\ has\text{-}disc\ disc2\ m \longrightarrow \neg\ has\text{-}disc\ disc2\ ms) \Longrightarrow matches\text{-}other \longleftrightarrow matches\ \gamma\ ms\ a\ p)$
$\Longrightarrow$
$matches\ \gamma\ (alist\text{-}and\ (NegPos\text{-}map\ C\ as))\ a\ p \wedge matches\text{-}other \longleftrightarrow matches\ \gamma\ m\ a\ p$
⟨*proof*⟩

lemma *primitive-extractor-matches-lastE*: $wf\text{-}disc\text{-}sel\ (disc,sel)\ C \Longrightarrow normalized\text{-}nnf\text{-}match\ m \Longrightarrow primitive\text{-}extractor\ (disc,\ sel)\ m = (as,\ ms)$
$\Longrightarrow$
$(normalized\text{-}nnf\text{-}match\ ms \Longrightarrow \neg\ has\text{-}disc\ disc\ ms \Longrightarrow (\forall\ disc2.\ \neg\ has\text{-}disc\ disc2\ m \longrightarrow \neg\ has\text{-}disc\ disc2\ ms) \Longrightarrow matches\ \gamma\ ms\ a\ p)$
$\Longrightarrow$
$matches\ \gamma\ (alist\text{-}and\ (NegPos\text{-}map\ C\ as))\ a\ p \longleftrightarrow matches\ \gamma\ m\ a\ p$
⟨*proof*⟩

The lemmas $[\![wf\text{-}disc\text{-}sel\ (?disc,\ ?sel)\ ?C;\ normalized\text{-}nnf\text{-}match\ ?m;\ primitive\text{-}extractor\ (?disc,\ ?sel)\ ?m = (?as,\ ?ms);\ [\![normalized\text{-}nnf\text{-}match\ ?ms;\ \neg\ has\text{-}disc\ ?disc\ ?ms;\ \forall\ disc2.\ \neg\ has\text{-}disc\ disc2\ ?m \longrightarrow \neg\ has\text{-}disc\ disc2\ ?ms]\!] \Longrightarrow ?matches\text{-}other = matches\ ?\gamma\ ?ms\ ?a\ ?p]\!] \Longrightarrow (matches\ ?\gamma\ (alist\text{-}and\ (NegPos\text{-}map\ ?C\ ?as))\ ?a\ ?p \wedge ?matches\text{-}other) = matches\ ?\gamma\ ?m\ ?a\ ?p$ and $[\![wf\text{-}disc\text{-}sel\ (?disc,\ ?sel)\ ?C;\ normalized\text{-}nnf\text{-}match\ ?m;\ primitive\text{-}extractor\ (?disc,\ ?sel)\ ?m = (?as,\ ?ms);\ [\![normalized\text{-}nnf\text{-}match\ ?ms;\ \neg\ has\text{-}disc\ ?disc\ ?ms;\ \forall\ disc2.\ \neg\ has\text{-}disc\ disc2\ ?m \longrightarrow \neg\ has\text{-}disc\ disc2\ ?ms]\!] \Longrightarrow matches\ ?\gamma\ ?ms\ ?a\ ?p]\!] \Longrightarrow matches\ ?\gamma\ (alist\text{-}and\ (NegPos\text{-}map\ ?C\ ?as))\ ?a\ ?p$

= *matches ?γ ?m ?a ?p* can be used as erule to solve goals about consecutive 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*⟩

thm *match-list-semantics*[*of* γ (*map* (*Match* ∘ *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*⟩

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*⟩

lemma *normalize-rules-primitive-extract-preserves-nnf-normalized*:
$\forall$ *r* $\in$ *set rs*. *normalized-nnf-match* (*get-match r*) $\Longrightarrow$ *wf-disc-sel disc-sel C* $\Longrightarrow$
$\forall$ *r* $\in$ *set* (*normalize-rules* (*normalize-primitive-extract disc-sel C f*) *rs*). *normalized-nnf-match* (*get-match r*)
⟨*proof*⟩

If something is normalized for disc2 and disc2 $\neq$ 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 $\forall$ *a*. $\neg$ *disc2* (*C a*) — disc1 and disc2 match for different stuff. e.g. *Src-Ports* and *Dst-Ports*
shows $\forall$ *mn* $\in$ *set* (*normalize-primitive-extract* (*disc1*, *sel1*) *C f m*). *normalized-n-primitive* (*disc2*, *sel2*) *P mn*
⟨*proof*⟩

lemma *normalize-primitive-extract-normalizes-n-primitive*:
fixes *disc*::('*a* $\Rightarrow$ *bool*) **and** *sel*::('*a* $\Rightarrow$ '*b*) **and** *f*::('*b negation-type list* $\Rightarrow$ '*b list*)
assumes *normalized-nnf-match m*
and *wf-disc-sel* (*disc*, *sel*) *C*
and *np*: $\forall$ *as*. ($\forall$ *a'* $\in$ *set* (*f as*). *P a'*)
shows $\forall$ *m'* $\in$ *set* (*normalize-primitive-extract* (*disc*, *sel*) *C f m*). *normalized-n-primitive* (*disc*, *sel*) *P m'*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *remove-unknowns-generic-not-has-disc*: $\neg$ *has-disc C m* $\Longrightarrow$ $\neg$ *has-disc C* (*remove-unknowns-generic* γ *a m*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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 $\exists$ *m-DNF*$\in$*set* (*normalize-match ?m*). *has-disc-negated ?disc ?neg m-DNF* $\Longrightarrow$ *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-da*
$\neg$ *has-disc-negated disc neg m* $\Longrightarrow$ $\forall$ *m-DNF* $\in$ *set* (*normalize-match m*). $\neg$ *has-disc-negated disc neg m-DNF*
⟨*proof*⟩

lemma *not-has-disc-opt-MatchAny-match-expr*:
$\neg$ *has-disc disc m* $\Longrightarrow$ $\neg$ *has-disc disc* (*opt-MatchAny-match-expr m*)
⟨*proof*⟩
lemma *not-has-disc-negated-opt-MatchAny-match-expr*:
$\neg$ *has-disc-negated disc neg m* $\Longrightarrow$ $\neg$ *has-disc-negated disc neg* (*opt-MatchAny-match-expr m*)

⟨*proof*⟩

lemma *normalize-match-preserves-nodisc*:
¬ *has-disc disc m* ⟹ *m′* ∈ *set* (*normalize-match m*) ⟹ ¬ *has-disc disc m′*
⟨*proof*⟩

lemma *not-has-disc-normalize-match*:
¬ *has-disc-negated disc neg m* ⟹ *m′* ∈ *set* (*normalize-match m*) ⟹ ¬ *has-disc-negated disc neg m′*
⟨*proof*⟩

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*
⟨*proof*⟩

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 ⟹ *compress-normalize-primitive disc-sel C f m* = *Some m′* ⟹
normalized-nnf-match m′
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *compress-normalize-primitive-hasdisc*:
assumes *am*: $\neg$ *has-disc disc2 m*
and *wf*: *wf-disc-sel* (*disc*,*sel*) *C*

and *disc*: $(\forall\, a.\ \neg\ disc2\ (C\ a))$
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*⟩

lemma *compress-normalize-primitive-hasdisc-negated*:
assumes *am*: ¬ *has-disc-negated disc2 neg m*
and *wf*: *wf-disc-sel* (*disc*,*sel*) *C*
and *disc*: $(\forall\, a.\ \neg\ disc2\ (C\ a))$
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*⟩

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*: $(\forall\, a.\ \neg\ disc2\ (C\ a))$
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*⟩

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*⟩

lemma *compress-normalize-primitive-monad-None*:
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 $\bigwedge m\ f.\ f \in set\ fs \Longrightarrow normalized\text{-}nnf\text{-}match\ m \Longrightarrow f\ m = None \Longrightarrow$ $\neg\ matches\ \gamma\ m\ a\ p$
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*
⟨*proof*⟩

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′* ∧ *P m′*
⟨*proof*⟩

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* ⇒ *′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*)
⟨*proof*⟩

lemma *andfold-MatchExp-matches*:
matches γ (*andfold-MatchExp ms*) *a p* ⟷ (∀ *m* ∈ *set ms*. *matches* γ *m a p*)
⟨*proof*⟩

lemma *andfold-MatchExp-not-discI*:
∀ *m* ∈ *set ms*. ¬ *has-disc disc m* ⟹ ¬ *has-disc disc* (*andfold-MatchExp ms*)
⟨*proof*⟩

lemma *andfold-MatchExp-not-disc-negatedI*:
∀ *m* ∈ *set ms*. ¬ *has-disc-negated disc neg m* ⟹ ¬ *has-disc-negated disc neg* (*andfold-MatchExp ms*)
⟨*proof*⟩

lemma *andfold-MatchExp-not-disc-negated-mapMatch*:
¬ *has-disc-negated disc False* (*andfold-MatchExp* (*map* (*Match* ∘ *C*) *ls*))
⟨*proof*⟩

lemma *andfold-MatchExp-not-disc-mapMatch*:
∀ *a*. ¬ *disc* (*C a*) ⟹ ¬ *has-disc disc* (*andfold-MatchExp* (*map* (*Match* ∘ *C*) *ls*))
⟨*proof*⟩

lemma *andfold-MatchExp-normalized-nnf*: ∀ *m* ∈ *set ms*. *normalized-nnf-match m* ⟹
normalized-nnf-match (*andfold-MatchExp ms*)
⟨*proof*⟩

lemma *andfold-MatchExp-normalized-n-primitive*: ∀ *m* ∈ *set ms*. *normalized-n-primitive* (*disc*, *sel*) *f m* ⟹
normalized-n-primitive (*disc*, *sel*) *f* (*andfold-MatchExp ms*)
⟨*proof*⟩

lemma *andfold-MatchExp-normalized-normalized-n-primitive-single*:
∀ *a*. ¬ *disc* (*C a*) ⟹
s ∈ *set* (*normalize-match* (*andfold-MatchExp* (*map* (*Match* ∘ *C*) *xs*))) ⟹
normalized-n-primitive (*disc*, *sel*) *f s*
⟨*proof*⟩

lemma *normalize-andfold-MatchExp-normalized-n-primitive*:
∀ *m* ∈ *set ms*. ∀ *s′* ∈ *set* (*normalize-match m*). *normalized-n-primitive* (*disc*, *sel*) *f s′* ⟹
s ∈ *set* (*normalize-match* (*andfold-MatchExp ms*)) ⟹
normalized-n-primitive (*disc*, *sel*) *f s*
⟨*proof*⟩
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* ⟷ *has-disc is-Extra m*
⟨*proof*⟩

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*
⟨*proof*⟩
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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *l4-ports-compress-length-Matchall*: *length pss* > *0* ⟹ *l4-ports-compress pss* ≠ *MatchesAll*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *l4-ports-negate-one-nodisc*:
∀ *a*. ¬ *disc* (*C a*) ⟹ ∀ *a*. ¬ *disc* (*Prot a*) ⟹ ¬ *has-disc disc* (*l4-ports-negate-one C pt*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

beware, the result is not nnf normalized!

lemma ¬ *normalized-nnf-match* (*l4-ports-negate-one C ports*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)]))] ⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*: ¬ *has-disc disc2 m*

and *negate-f*: *has-disc-negated disc False m* ⟹ ∀ *pts*. ¬ *has-disc disc2* (*negate-f C pts*)
and *no-disc*: ∀ *a*. ¬ *disc2* (*C a*)
shows ¬ *has-disc disc2* (*rewrite-negated-primitives* (*disc*,*sel*) *C negate-f m*)

⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

definition *rewrite-negated-src-ports*
:: *'i*::*len common-primitive match-expr* $\Rightarrow$ *'i common-primitive match-expr*
where
rewrite-negated-src-ports m $\equiv$
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* $\Rightarrow$ *'i common-primitive match-expr*
where
rewrite-negated-dst-ports m $\equiv$
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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma $\neg$ *has-disc-negated disc t m* $\Longrightarrow$ $\forall m' \in$ *set* (*normalize-match m*). $\neg$ *has-disc-negated disc t m'*
⟨*proof*⟩

corollary *normalize-rewrite-negated-src-ports-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m*
shows $\forall m' \in$ *set* (*normalize-match* (*rewrite-negated-src-ports m*)). $\neg$ *has-disc-negated is-Src-Ports False m'*
⟨*proof*⟩

27.4 Normalizing Positive Matches on Ports

fun *singletonize-L4Ports* :: *ipt-l4-ports* $\Rightarrow$ *ipt-l4-ports list* **where**
singletonize-L4Ports (*L4Ports proto pts*) = *map* (λp. *L4Ports proto* [*p*]) *pts*

lemma *singletonize-L4Ports-src*: **assumes** *generic*: *primitive-matcher-generic* β
shows *match-list* (β, α) (*map* (*Match* $\circ$ *Src-Ports*) (*singletonize-L4Ports pts*))

a p ⟷
matches (β, α) (*Match* (*Src-Ports pts*)) *a p*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *singletonize-L4Ports-normalized-generic*:
assumes *wf-disc-sel*: *wf-disc-sel* (*disc*,*sel*) *C*
and *m′* ∈ (λ*spt*. *Match* (*C spt*)) ‘ *set* (*singletonize-L4Ports pt*)
shows *normalized-n-primitive* (*disc*, *sel*) (*case-ipt-l4-ports* (λ*x pts*. *length pts* ≤ *1*)) *m′*
⟨*proof*⟩

lemma *singletonize-L4Ports-normalized-src-ports*:
m′ ∈ (λ*spt*. *Match* (*Src-Ports spt*)) ‘ *set* (*singletonize-L4Ports pt*) ⟹ *normalized-src-ports m′*
⟨*proof*⟩

lemma *singletonize-L4Ports-normalized-dst-ports*:
m′ ∈ (λ*spt*. *Match* (*Dst-Ports spt*)) ‘ *set* (*singletonize-L4Ports pt*) ⟹ *normalized-dst-ports m′*
⟨*proof*⟩

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* (λ*x pts*. *length pts* ≤ *1*)) *rst* ⟹
m′ ∈ (λ*spt*. *MatchAnd* (*Match* (*C spt*)) *rst*) ‘ *set* (*singletonize-L4Ports pt*) ⟹
normalized-n-primitive (*disc*, *sel*) (*case-ipt-l4-ports* (λ*x pts*. *length pts* ≤ *1*)) *m′*
⟨*proof*⟩

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′*
⟨*proof*⟩

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*) (*λps. case ps of L4Ports - pts* ⇒ *length pts* ≤ *1*) *m′*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟷ *matches* (*β*, *α*) *m a p*
⟨*proof*⟩

lemma *normalize-positive-dst-ports*:
assumes *generic*: *primitive-matcher-generic β*

and *n*: *normalized-nnf-match m*
and *noneg*: ¬ *has-disc-negated is-Dst-Ports False m*
shows *match-list* (β, α) (*normalize-positive-dst-ports m*) *a p* ⟷ *matches* (β, α) *m a p*
⟨*proof*⟩

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'*
⟨*proof*⟩
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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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 ∀ *m'* ∈ *set* (*normalize-positive-dst-ports m*). *normalized-dst-ports m'*
⟨*proof*⟩

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'* ∈ *set* (*normalize-ports-generic normalize-pos rewrite-neg m*)
and *noNeg*: ¬ *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'*

shows *normalized-nnf-match m'*
⟨*proof*⟩

lemma *normalize-ports-generic*:
assumes *n*: *normalized-nnf-match m*
and *normalize-pos*: $\bigwedge m$. *normalized-nnf-match m* $\Longrightarrow \neg$ *has-disc-negated disc False m* $\Longrightarrow$
match-list γ (*normalize-pos m*) *a p* $\longleftrightarrow$ *matches* γ *m a p*
and *rewrite-neg*: $\bigwedge m$. *normalized-nnf-match m* $\Longrightarrow$
matches γ (*rewrite-neg m*) *a p* = *matches* γ *m a p*
and *noNeg*: $\bigwedge m$. *normalized-nnf-match m* $\Longrightarrow \neg$ *has-disc-negated disc False* (*rewrite-neg m*)
shows
match-list γ (*normalize-ports-generic normalize-pos rewrite-neg m*) *a p* $\longleftrightarrow$ *matches* γ *m a p*
⟨*proof*⟩

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*: $\bigwedge m$. *normalized-nnf-match m* $\Longrightarrow \neg$ *has-disc-negated disc False* (*rewrite-neg m*)
and *normalize-nnf-pos*: $\bigwedge m\ m'$.
normalized-nnf-match m $\Longrightarrow \neg$ *has-disc-negated disc False m* $\Longrightarrow$
$m' \in$ *set* (*normalize-pos m*) $\Longrightarrow$ *normalized-nnf-match m'*
and *normalize-pos*: $\bigwedge m\ m'$.
normalized-nnf-match m $\Longrightarrow \neg$ *has-disc-negated disc False m* $\Longrightarrow$
$\forall\, m' \in$ *set* (*normalize-pos m*).
normalized-n-primitive (*disc*,*sel*) (λ*ps*. *case ps of L4Ports - pts* $\Rightarrow$ *length pts* $\leq$ *1*) *m'*
shows $\forall\, m' \in$ *set* (*normalize-ports-generic normalize-pos rewrite-neg m*).
normalized-n-primitive (*disc*,*sel*) (λ*ps*. *case ps of L4Ports - pts* $\Rightarrow$ *length pts* $\leq$ *1*) *m'*
⟨*proof*⟩

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*: $\forall\, a$. $\neg$ *disc* (*Prot a*)
and *P*: *P* (*disc2*, *sel2*) *m*
and *P1*: $\bigwedge a$. *normalized-nnf-match a* $\Longrightarrow$
a $\in$ *set* (*normalize-match* (*rewrite-negated-primitives* (*disc*, *sel*) *C l4-ports-negate-one m*)) $\Longrightarrow$
P (*disc2*, *sel2*) *a*
and *P2*: $\bigwedge$*a dpts rst*. *normalized-nnf-match a* $\Longrightarrow$
primitive-extractor (*disc*, *sel*) *a* = (*dpts*, *rst*) $\Longrightarrow$
getNeg dpts = [] $\Longrightarrow$ *P* (*disc2*, *sel2*) *a* $\Longrightarrow$ *P* (*disc2*, *sel2*) *rst*
and *P3*: $\bigwedge$ *a spt rst*. *P* (*disc2*, *sel2*) *rst* $\Longrightarrow$ *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′*
⟨*proof*⟩

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*
⟨*proof*⟩

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*: ∀ *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*)
⟹ ¬ *has-disc disc2 m′*
⟨*proof*⟩

lemma *normalize-ports-generic-preserves-normalized-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m* **and** *nodisc*: ¬ *has-disc-negated disc2 False m*
and *wf-disc-sel*: *wf-disc-sel* (*disc*, *sel*) *C*
and *noProt*: ∀ *a*. ¬ *disc* (*Prot 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*)
⟹ ¬ *has-disc-negated disc2 False m′*
⟨*proof*⟩

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* $\longleftrightarrow$ *matches* (β, α) *m a p*
⟨*proof*⟩

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* $\longleftrightarrow$ *matches* (β, α) *m a p*
⟨*proof*⟩

lemma *normalize-src-ports-normalized-n-primitive*:
assumes *n*:*normalized-nnf-match m*
shows $\forall$ *m'* $\in$ *set* (*normalize-src-ports m*). *normalized-src-ports m'*
⟨*proof*⟩

lemma *normalize-dst-ports-normalized-n-primitive*:
assumes *n*: *normalized-nnf-match m*
shows $\forall$ *m'* $\in$ *set* (*normalize-dst-ports m*). *normalized-dst-ports m'*
⟨*proof*⟩

lemma *normalize-src-ports-nnf*:
assumes *n*: *normalized-nnf-match m*
shows *m'* $\in$ *set* (*normalize-src-ports m*) $\Longrightarrow$ *normalized-nnf-match m'*
⟨*proof*⟩

lemma *normalize-dst-ports-nnf*:
assumes *n*: *normalized-nnf-match m*
shows *m'* $\in$ *set* (*normalize-dst-ports m*) $\Longrightarrow$ *normalized-nnf-match m'*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *normalize-dst-ports-preserves-normalized-n-primitive*:
assumes *n*: *normalized-nnf-match 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*) ⟹
normalized-n-primitive (*disc2*, *sel2*) *f* *m* ⟹
normalized-n-primitive (*disc2*, *sel2*) *f m'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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*))))] ⟨*proof*⟩

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*)))))] ⟨*proof*⟩

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*)))] ⟨*proof*⟩

lemma *normalize-match* (*andfold-MatchExp* (*map* (*l4-ports-negate-one C*) [])) = [*MatchAny*] ⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟷

matches (β, α) *m a p*
⟨*proof*⟩

lemma ∀ *a*. ¬ *disc* (*Src-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Dst-Ports a*) ⟹
normalized-n-primitive (*disc*, *sel*) *f m* ⟹
∀ *m'* ∈ *set* (*normalize-match* (*rewrite-MultiportPorts-one a*)).
normalized-n-primitive (*disc*, *sel*) *f m'*
⟨*proof*⟩

lemma *rewrite-MultiportPorts-one-nodisc*:
∀ *a*. ¬ *disc* (*Src-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Dst-Ports a*) ⟹
¬ *has-disc disc* (*rewrite-MultiportPorts-one a*)
∀ *a*. ¬ *disc* (*Src-Ports a*) ⟹ ∀ *a*. ¬ *disc* (*Dst-Ports a*) ⟹
¬ *has-disc-negated disc neg* (*rewrite-MultiportPorts-one a*)
⟨*proof*⟩

definition *rewrite-MultiportPorts*
:: *'i::len common-primitive match-expr* ⇒ *'i common-primitive match-expr list*
where
rewrite-MultiportPorts m ≡ *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* ⟷ *matches* (β, α) *m a p*
⟨*proof*⟩

lemma *rewrite-MultiportPorts-normalized-nnf-match*:
m' ∈ *set* (*rewrite-MultiportPorts m*) ⟹ *normalized-nnf-match m'*
⟨*proof*⟩

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*]
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *rewrite-MultiportPorts-removes-MultiportsPorts*:
assumes *n*: *normalized-nnf-match m*
shows *m'* ∈ *set* (*rewrite-MultiportPorts m*) ⟹ ¬ *has-disc is-MultiportPorts m'*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩
lemma *normalize-src-ips*: *normalized-nnf-match m* ⟹

match-list (*common-matcher*, α) (*normalize-src-ips m*) *a p* = *matches* (*common-matcher*, α) *m a p*
⟨*proof*⟩

lemma *normalize-src-ips-normalized-n-primitive*: *normalized-nnf-match m* $\Longrightarrow$
$\forall$ *m'* $\in$ *set* (*normalize-src-ips m*). *normalized-src-ips m'*
⟨*proof*⟩

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*⟩
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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *normalize-dst-ips-normalized-n-primitive*: *normalized-nnf-match m* $\Longrightarrow$
$\forall$ *m'* $\in$ *set* (*normalize-dst-ips m*). *normalized-dst-ips m'*
⟨*proof*⟩

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
  ¬ *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* $\Longrightarrow$ ¬ *matches* (β, α) (*alist-and* (*NegPos-map IIface ifces*)) *a p*
compress-interfaces ifces = *None* $\Longrightarrow$ ¬ *matches* (β, α) (*alist-and* (*NegPos-map OIface ifces*)) *a p*
⟨*proof*⟩ **lemma** *compress-interfaces-Some*:
assumes *generic*: *primitive-matcher-generic* β
shows
compress-interfaces ifces = *Some* (*i-pos*, *i-neg*) $\Longrightarrow$
matches (β, α) (*alist-and* (*NegPos-map IIface* ((*map Pos i-pos*)@(*map Neg i-neg*)))) *a p* $\longleftrightarrow$
matches (β, α) (*alist-and* (*NegPos-map IIface ifces*)) *a p*
compress-interfaces ifces = *Some* (*i-pos*, *i-neg*) $\Longrightarrow$
matches (β, α) (*alist-and* (*NegPos-map OIface* ((*map Pos i-pos*)@(*map Neg i-neg*)))) *a p* $\longleftrightarrow$
matches (β, α) (*alist-and* (*NegPos-map OIface ifces*)) *a p*
⟨*proof*⟩

definition *compress-normalize-input-interfaces* :: *'i*::*len common-primitive match-expr* $\Rightarrow$ *'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* $\longleftrightarrow$ *matches* (β, α) *m a p*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *compress-normalize-input-interfaces-nnf*: *normalized-nnf-match m* $\Longrightarrow$ *compress-normalize-input-interfaces m* = *Some m'* $\Longrightarrow$
normalized-nnf-match m'
⟨*proof*⟩

lemma *compress-normalize-input-interfaces-not-introduces-Iiface*:
¬ *has-disc is-Iiface m* $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *compress-normalize-input-interfaces m* = *Some m'* $\Longrightarrow$
¬ *has-disc is-Iiface m'*
⟨*proof*⟩

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-Iiface False m'*
⟨*proof*⟩

lemma *compress-normalize-input-interfaces-hasdisc*:
¬ *has-disc disc m* $\Longrightarrow$ (∀ *a*. ¬ *disc* (*IIface a*)) $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *compress-normalize-input-interfaces m* = *Some m'* $\Longrightarrow$
normalized-nnf-match m' ∧ ¬ *has-disc disc m'*
⟨*proof*⟩

lemma *compress-normalize-input-interfaces-hasdisc-negated*:
¬ *has-disc-negated disc neg m* $\Longrightarrow$ (∀ *a*. ¬ *disc* (*IIface a*)) $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *compress-normalize-input-interfaces m* = *Some m'* $\Longrightarrow$
normalized-nnf-match m' ∧ ¬ *has-disc-negated disc neg m'*
⟨*proof*⟩

lemma *compress-normalize-input-interfaces-preserves-normalized-n-primitive*:
normalized-n-primitive (*disc*, *sel*) *P m* $\Longrightarrow$ (∀ *a*. ¬ *disc* (*IIface a*)) $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *compress-normalize-input-interfaces m* = *Some m'* $\Longrightarrow$
normalized-nnf-match m' ∧ *normalized-n-primitive* (*disc*, *sel*) *P m'*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *compress-normalize-output-interfaces-nnf*: *normalized-nnf-match m* ⟹ *compress-normalize-output-interfaces m* = *Some m'* ⟹
normalized-nnf-match m'
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *compress-normalize-output-interfaces-hasdisc*:
¬ *has-disc disc m* ⟹ (∀ *a*. ¬ *disc* (*OIface a*)) ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-output-interfaces m* = *Some m'* ⟹
normalized-nnf-match m' ∧ ¬ *has-disc disc m'*
⟨*proof*⟩

lemma *compress-normalize-output-interfaces-hasdisc-negated*:
¬ *has-disc-negated disc neg m* ⟹ (∀ *a*. ¬ *disc* (*OIface a*)) ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-output-interfaces m* = *Some m′* ⟹
normalized-nnf-match m′ ∧ ¬ *has-disc-negated disc neg m′*
⟨*proof*⟩

lemma *compress-normalize-output-interfaces-preserves-normalized-n-primitive*:
normalized-n-primitive (*disc*, *sel*) *P m* ⟹ (∀ *a*. ¬ *disc* (*OIface a*)) ⟹ *normalized-nnf-match m* ⟹ *compress-normalize-output-interfaces m* = *Some m′* ⟹
normalized-nnf-match m′ ∧ *normalized-n-primitive* (*disc*, *sel*) *P m′*
⟨*proof*⟩

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* ⇒ ′*a word* ⇒ ′*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*⟩

lemma *word-interval-eq-set-word-range* [*code-unfold*]:
‹{*v*..*w*} = *set* (*word-range v w*)›
⟨*proof*⟩

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*])
⟨*proof*⟩

termination *word-upto*
⟨*proof*⟩

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*)
⟨*proof*⟩

termination *word-upto′*
⟨*proof*⟩

declare *word-upto′.simps*[*simp del*]

lemma *word-upto-cons-front*[*code*]:
word-upto a b = *word-upto′ a b*
⟨*proof*⟩

lemma *word-upto-set-eq*: $a \le b \Longrightarrow x \in set\ (word\text{-}upto\ a\ b) \longleftrightarrow a \le x \wedge x \le b$
⟨*proof*⟩

lemma *word-upto-distinct-hlp*: $a \le b \Longrightarrow a \ne b \Longrightarrow b \notin set\ (word\text{-}upto\ a\ (b - 1))$
⟨*proof*⟩

lemma *distinct-word-upto*: $a \le b \Longrightarrow distinct\ (word\text{-}upto\ a\ b)$
⟨*proof*⟩

lemma *word-upto-eq-upto*: $s \le e \Longrightarrow e \le unat\ (max\text{-}word :: 'l\ word) \Longrightarrow$
word-upto ((*of-nat* :: *nat* ⇒ (*′l* :: *len*) *word*) *s*) (*of-nat e*) = *map of-nat* (*upt s* (*Suc e*))
⟨*proof*⟩

lemma *word-upto-alt*: $(a :: ('l :: len)\ word) \le b \Longrightarrow$
word-upto a b = *map of-nat* (*upt* (*unat a*) (*Suc* (*unat b*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *sorted-word-upto*:
fixes *a b* :: (*′l* :: *len*) *word*
assumes *a* ≤ *b*
shows *sorted* (*word-upto a b*)

⟨*proof*⟩

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*⟩

lemma *compress-pos-protocols-None*: *compress-pos-protocols ps* = *None* ⟹
¬ (∀ *proto* ∈ *set ps*. *match-proto proto p-prot*)
⟨*proof*⟩

lemma *simple-proto-conjunct* (*Proto p1*) (*Proto p2*) ≠ *None* ⟹ ∀ *pkt*. *match-proto* (*Proto p1*) *pkt* ⟷ *match-proto* (*Proto p2*) *pkt*
⟨*proof*⟩
lemma *simple-proto-conjunct p1* (*Proto p2*) ≠ *None* ⟹ ∀ *pkt*. *match-proto* (*Proto p2*) *pkt* ⟶ *match-proto p1 pkt*
⟨*proof*⟩

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*⟩

lemma *set-word8-word-upto*: {*0*..(− *1* :: *8 word*)} = *set* (*word-upto 0 255*)
⟨*proof*⟩
lemma (∀ *p* ∈ {*0*..− *1*}. *Proto p* ∈ *set* (*getNeg ps*)) ⟷
((∀ *p* ∈ *set* (*word-upto 0 255*). *Proto p* ∈ *set* (*getNeg ps*)))
⟨*proof*⟩

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*], [])
)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

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*⟩

lemma *compress-normalize-protocols-step-nnf*:
normalized-nnf-match m ⟹ *compress-normalize-protocols-step m = Some m′* ⟹
normalized-nnf-match m′
⟨*proof*⟩

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′*
⟨*proof*⟩

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′*
⟨*proof*⟩

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′*
⟨*proof*⟩

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′*
⟨*proof*⟩

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′*
⟨*proof*⟩

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*''))) ⟨*proof*⟩

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* (λ*proto 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*
⟨*proof*⟩

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* (λ*proto x*. *Proto proto*))))
(*getPos srcpts*)))
(*andfold-MatchExp*
(*map* (*Match* ∘ (*Prot* ∘ (*case-ipt-l4-ports* (λ*proto x*. *Proto proto*))))
(*getPos dstpts*))))
(*alist-and′* (*NegPos-map Src-Ports srcpts* @ *NegPos-map Dst-Ports dstpts*)))
rst2)) ⟹
P (*import-protocols-from-ports m*)
⟨*proof*⟩

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*⟩

lemma *import-protocols-from-ports-nnf*:
normalized-nnf-match m ⟹ *normalized-nnf-match* (*import-protocols-from-ports m*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *import-protocols-from-ports-preserves-normalized-n-primitive*:
normalized-n-primitive (*disc*, *sel*) *f m* $\Longrightarrow$ ($\forall$ *a*. ¬ *disc* (*Prot a*)) $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$
normalized-nnf-match (*import-protocols-from-ports m*) $\wedge$ *normalized-n-primitive* (*disc*, *sel*) *f* (*import-protocols-from-ports m*)
⟨*proof*⟩

30.2 Putting things together

definition *compress-normalize-protocols*
:: *'i::len common-primitive match-expr* $\Rightarrow$ *'i common-primitive match-expr option* **where**
compress-normalize-protocols m $\equiv$ *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* $\longleftrightarrow$ *matches* (β, α) *m a p*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *compress-normalize-protocols-nnf*:
normalized-nnf-match m $\Longrightarrow$ *compress-normalize-protocols m* = *Some m'* $\Longrightarrow$
normalized-nnf-match m'
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *compress-normalize-protocols-hasdisc*:
¬ *has-disc disc m* $\Longrightarrow$ ($\forall$ *a*. ¬ *disc* (*Prot a*)) $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *compress-normalize-protocols m* = *Some m'* $\Longrightarrow$
normalized-nnf-match m' $\wedge$ ¬ *has-disc disc m'*
⟨*proof*⟩

lemma *compress-normalize-protocols-hasdisc-negated*:
¬ *has-disc-negated disc False m* $\Longrightarrow$ ($\forall$ *a*. ¬ *disc* (*Prot a*)) $\Longrightarrow$

normalized-nnf-match m ⟹ *compress-normalize-protocols m* = *Some m′* ⟹
normalized-nnf-match m′ ∧ ¬ *has-disc-negated disc False m′*
⟨*proof*⟩

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′*
⟨*proof*⟩

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*′′))) ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *remdups-rev-append*: *remdups-rev* (*rs* @ *rs2*) = *remdups-rev rs* @ *remdups-rev* [*r*←*rs2* . *r* ∉ *set rs*]
⟨*proof*⟩

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*⟩

lemma *remdups-rev-set*: *set* (*remdups-rev rs*) = *set rs* ⟨*proof*⟩

lemma *remdups-rev-removeAll*: *remdups-rev* (*removeAll r rs*) = *removeAll r* (*remdups-rev rs*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *ipassmt-sanity-nowildcards-match-iface*:
ipassmt-sanity-nowildcards ipassmt ⟹
ipassmt (*Iface ifce2*) = *None* ⟹
ipassmt ifce = *Some a* ⟹
¬ *match-iface ifce ifce2*
⟨*proof*⟩

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-must-be-distinct-and-dont-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*)))))))
⟨*proof*⟩

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*
)
⟨*proof*⟩

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* (*λx. 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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)])]) ⟨*proof*⟩

definition *ipassmt-ignore-wildcard* :: *′i::len ipassignment* ⇒ *′i ipassignment* **where**
ipassmt-ignore-wildcard ipassmt ≡ *λk. 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* $\subseteq_m$ *ipassmt*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))) = {})
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *ipassmt-ignore-wildcard-None-Some*:
ipassmt-ignore-wildcard ipassmt ifce = *None* ⟹ *ipassmt ifce* = *Some ips* ⟹ *ipcidr-union-set* (*set ips*) = *UNIV*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *ipcidr-union-cidr-split*[*simp*]: *ipcidr-union-set* (*set* (*cidr-split a*)) = *wordinterval-to-set a*
⟨*proof*⟩

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*
⟨*proof*⟩

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)])] ⟨proof⟩

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*⟩ $\Rightarrow_\alpha$ *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*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow*)
⟶ *p-src p* ∈ (*ipcidr-union-set* (*set* (*the* (*ipassmt iface*)))))
⟨proof⟩

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-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*, -) = (*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''*)))) ⟨*proof*⟩ **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*⟩

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*
⟨*proof*⟩

lemma ¬ *has-primitive m* ∧ *matcheq-matchAny m* ⟷ (*if* ¬ *has-primitive m then matcheq-matchAny m else False*)
⟨*proof*⟩

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*⟩ **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-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*, -) = (*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*
⟨*proof*⟩

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*) ⟨*proof*⟩ **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*
⟨*proof*⟩
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*) ⟨*proof*⟩

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 ipcidr-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*⟩ **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*⟩ **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*
⟨*proof*⟩ **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*
⟨*proof*⟩ **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*}
⟨*proof*⟩ **lemma** *not-FinalAllow*: *foo* ≠ *Decision FinalAllow* ⟷ *foo* = *Decision FinalDeny* ∨ *foo* = *Undecided*
⟨*proof*⟩ **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

⟨*proof*⟩ **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*

⟨*proof*⟩ **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*

⟨*proof*⟩ **lemma** *Collect-minus-eq*: {*x*. *P x*} − {*x*. *Q x*} = {*x*. *P x* ∧ ¬ *Q x*}

⟨*proof*⟩ **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*

⟨*proof*⟩ **lemma** *setbydecision-all TYPE*('*pkt-ext*) *iface rs FinalDeny* ⊆ − *setbydecision TYPE*('*pkt-ext*) *iface rs FinalAllow*

⟨*proof*⟩ **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*⟩

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

⟨*proof*⟩ **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*
⟨*proof*⟩

The *nospoof* definition used throughout the proofs corresponds to checking *no-spoofing* for all interfaces

private lemma *nospoof*: *simple-ruleset rs* ⟹ (∀ *iface* ∈ *dom ipassmt*. *nospoof TYPE*(*'pkt-ext*) *iface ipassmt rs*) ⟷ *no-spoofing TYPE*(*'pkt-ext*) *ipassmt rs*
⟨*proof*⟩

theorem *no-spoofing-iface*: *simple-ruleset rs* ⟹ ∀ *r*∈*set rs*. *normalized-nnf-match* (*get-match r*) ⟹
∀ *iface* ∈ *dom ipassmt*. *no-spoofing-iface iface ipassmt rs* ⟹ *no-spoofing TYPE*(*'pkt-ext*) *ipassmt rs*
⟨*proof*⟩

Examples

Example 1: Ruleset: Accept all non-spoofed packets, drop rest.

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*))) (*Match* (*IIface* (*Iface ''eth0''*)))) *action.Accept*,
Rule MatchAny action.Drop] ⟨*proof*⟩
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*))) (*Match* (*IIface* (*Iface ''eth0''*)))) *action.Accept*,
Rule MatchAny action.Drop]
⟨*proof*⟩

Example 2: Ruleset: Drop packets from a spoofed IP range, allow rest. Handles negated interfaces correctly.

lemma *no-spoofing TYPE*(*'pkt-ext*)
[*Iface ''eth0''* ↦ [(*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]

⟨*proof*⟩

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]

⟨*proof*⟩

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]
⟨*proof*⟩

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]
⟨*proof*⟩

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] ⟨*proof*⟩

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* ∪ ¬*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*⟩

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] ⟨*proof*⟩

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] ⟨*proof*⟩

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*] ⟨*proof*⟩

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*]'' ⟨*proof*⟩

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-toSt 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*
⟨*proof*⟩ **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*)
⟨*proof*⟩

lemma *routing-ipassmt-wi* ([]::*32 prefix-routing*) = [(*output-iface* (*routing-action* (*undefined* :: *32 routing-rule*)), *WordInterval 0 0xFFFFFFFF*)]
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *routing-ipassmt-ipassmt-sanity-disjoint*: *valid-prefixes* (*rt*::(*'i*::*len*) *prefix-routing*) ⟹
ipassmt-sanity-disjoint (*map-of* (*routing-ipassmt rt*))
⟨*proof*⟩

lemma *routing-ipassmt-distinct*: *distinct* (*map fst* (*routing-ipassmt rtbl*))
⟨*proof*⟩

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*⟩

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*⟩

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*⟩ **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* ∧ *p-dst p* ∈ *ipcidr-union-set* (*set p-ips*)
shows *matches* (*common-matcher*, α) (*ipassmt-iface-replace-dstip-mexpr ipassmt ifce*) *a p* ⟷
matches (*common-matcher*, α) (*Match* (*OIface ifce*)) *a p*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩
end

lemma *oiface-rewrite-preserves-nodisc*:
∀ *a*. ¬ *disc* (*Dst a*) ⟹ ¬ *has-disc disc m* ⟹ ¬ *has-disc disc* (*oiface-rewrite*

ipassmt m)
⟨*proof*⟩

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* ⇒ *iface* ⇒ *'i common-primitive match-expr*
where
ipassmt-iface-constrain-srcip-mexpr ipassmt ifce = (*case ipassmt ifce of*
None ⇒ *Match* (*IIface ifce*)
| *Some ips* ⇒ *MatchAnd*
(*Match* (*IIface ifce*))
(*match-list-to-match-expr* (*map* (*Match* ∘ *Src*) (*map* (*uncurry IpAddrNetmask*) *ips*)))
)

lemma *matches-ipassmt-iface-constrain-srcip-mexpr*:
matches (*common-matcher*, α) (*ipassmt-iface-constrain-srcip-mexpr ipassmt ifce*) *a p* ⟷
(*case ipassmt ifce of*
None ⇒ *match-iface ifce* (*p-iiface p*)
| *Some ips* ⇒ *match-iface ifce* (*p-iiface p*) ∧ *p-src p* ∈ *ipcidr-union-set* (*set ips*)
)
⟨*proof*⟩

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*⟩ **lemma** *matches-ipassmt-iface-constrain-srcip-mexpr-case-Iface*:
fixes *ifce*::*iface*
assumes *ipassmt-sanity-nowildcards ipassmt*
and $\bigwedge$*ips. ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* $\Longrightarrow$ *p-src p* $\in$ *ipcidr-union-set* (*set ips*)
shows *matches* (*common-matcher*, α) (*ipassmt-iface-constrain-srcip-mexpr ipassmt ifce*) *a p* $\longleftrightarrow$
matches (*common-matcher*, α) (*Match* (*IIface ifce*)) *a p*
⟨*proof*⟩

lemma *matches-iiface-constrain*:
normalized-nnf-match m $\Longrightarrow$ *ipassmt-sanity-nowildcards ipassmt* $\Longrightarrow$
($\bigwedge$*ips. ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* $\Longrightarrow$ *p-src p* $\in$ *ipcidr-union-set* (*set ips*)) $\Longrightarrow$
matches (*common-matcher*, α) (*iiface-constrain ipassmt m*) *a p* $\longleftrightarrow$ *matches* (*common-matcher*, α) *m a p*
⟨*proof*⟩
end

36.2 Sanity checking the assumption

lemma ($\exists$ *ips. ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* $\wedge$ *p-src p* $\in$ *ipcidr-union-set* (*set ips*)) $\Longrightarrow$
(*case ipassmt* (*Iface* (*p-iiface p*)) *of Some ips* $\Rightarrow$ *p-src p* $\in$ *ipcidr-union-set* (*set ips*))
(*case ipassmt* (*Iface* (*p-iiface p*)) *of Some ips* $\Rightarrow$ *p-src p* $\in$ *ipcidr-union-set* (*set ips*)) $\Longrightarrow$
($\bigwedge$*ips. ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* $\Longrightarrow$ *p-src p* $\in$ *ipcidr-union-set* (*set ips*))
⟨*proof*⟩

Sanity check: If we assume that there are no spoofed packets, spoofing protection is trivially fulfilled.

lemma $\forall$ *p*:: (*'i*::*len*,*'pkt-ext*) *tagged-packet-scheme*.
Iface (*p-iiface p*) $\in$ *dom ipassmt* $\longrightarrow$ *p-src p* $\in$ *ipcidr-union-set* (*set* (*the*

(*ipassmt* (*Iface* (*p-iiface p*))))) $\Longrightarrow$
no-spoofing TYPE('pkt-ext) ipassmt rs
⟨*proof*⟩

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* $\Longrightarrow$
(*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow* $\Longrightarrow$
case ipassmt (*Iface* (*p-iiface p*)) *of Some ips* ⇒ *p-src p* ∈ *ipcidr-union-set* (*set ips*) | *None* ⇒ *True*
⟨*proof*⟩

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*⟩

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* $\longleftrightarrow$
matches (*common-matcher*, α) (*iiface-rewrite ipassmt* (*Match x*)) *a p* = *matches* (*common-matcher*, α) (*Match x*) *a p*
⟨*proof*⟩ **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* $\wedge$ *p-src p* $\in$ *ipcidr-union-set* (*set p-ips*)
shows *matches* (*common-matcher*, α) (*ipassmt-iface-replace-srcip-mexpr ipassmt ifce*) *a p* $\longleftrightarrow$
matches (*common-matcher*, α) (*Match* (*IIface ifce*)) *a p*
⟨*proof*⟩

lemma *matches-iiface-rewrite*:
normalized-nnf-match m $\Longrightarrow$ *ipassmt-sanity-nowildcards ipassmt* $\Longrightarrow$ *ipassmt-sanity-disjoint ipassmt* $\Longrightarrow$
($\exists$ *p-ips*. *ipassmt* (*Iface* (*p-iiface p*)) = *Some p-ips* $\wedge$ *p-src p* $\in$ *ipcidr-union-set* (*set p-ips*)) $\Longrightarrow$
matches (*common-matcher*, α) (*iiface-rewrite ipassmt m*) *a p* $\longleftrightarrow$ *matches* (*common-matcher*, α) *m a p*
⟨*proof*⟩

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*: $\bigwedge$ *ifce p*:: *'i*::*len tagged-packet*.
(*matches* (*common-matcher*, α) (*ipassmt-iface-replace-srcip-mexpr ipassmt ifce*) *a p* $\longleftrightarrow$ *matches* (*common-matcher*, α) (*Match* (*IIface ifce*)) *a p*)
shows *ipassmt-sanity-disjoint ipassmt*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

lemma *cut-off-after-match-any-simplers*: *simple-ruleset rs* ⟹ *simple-ruleset* (*cut-off-after-match-any* *rs*)

⟨*proof*⟩

lemma *cut-off-after-match-any-preserve-matches*:
∀ *r* ∈ *set rs*. *P* (*get-match r*) ⟹ ∀ *r* ∈ *set* (*cut-off-after-match-any rs*). *P* (*get-match r*)
⟨*proof*⟩

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* ⇒ *'i common-primitive match-expr option*
where
compress-normalize-besteffort m ≡ *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'*
⟨*proof*⟩ **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] $\Longrightarrow$
normalized-nnf-match m $\Longrightarrow$
f m = *Some m'* $\Longrightarrow$
matches (β, α) *m' a p* = *matches* (β, α) *m a p*
⟨*proof*⟩

lemma *compress-normalize-besteffort-Some*:
assumes *generic*: *primitive-matcher-generic* β
shows *normalized-nnf-match m* $\Longrightarrow$
compress-normalize-besteffort m = *Some m'* $\Longrightarrow$
matches (β, α) *m' a p* = *matches* (β, α) *m a p*
⟨*proof*⟩
lemma *compress-normalize-besteffort-None*:
assumes *generic*: *primitive-matcher-generic* β
shows *normalized-nnf-match m* $\Longrightarrow$
compress-normalize-besteffort m = *None* $\Longrightarrow$
¬ *matches* (β, α) *m a p*
⟨*proof*⟩
lemma *compress-normalize-besteffort-nnf*:
normalized-nnf-match m $\Longrightarrow$
compress-normalize-besteffort m = *Some m'* $\Longrightarrow$
normalized-nnf-match m'
⟨*proof*⟩

lemma *compress-normalize-besteffort-not-introduces-Iiface*:
¬ *has-disc is-Iiface m* $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *compress-normalize-besteffort m* = *Some m'* $\Longrightarrow$
¬ *has-disc is-Iiface m'*
⟨*proof*⟩
lemma *compress-normalize-besteffort-not-introduces-Oiface*:
¬ *has-disc is-Oiface m* $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *compress-normalize-besteffort m* = *Some m'* $\Longrightarrow$
¬ *has-disc is-Oiface m'*
⟨*proof*⟩

lemma *compress-normalize-besteffort-not-introduces-Iiface-negated*:
¬ *has-disc-negated is-Iiface False m* $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *compress-normalize-besteffort m* = *Some m'* $\Longrightarrow$
¬ *has-disc-negated is-Iiface False m'*
⟨*proof*⟩
lemma *compress-normalize-besteffort-not-introduces-Oiface-negated*:
¬ *has-disc-negated is-Oiface False m* $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *compress-normalize-besteffort m* = *Some m'* $\Longrightarrow$
¬ *has-disc-negated is-Oiface False m'*
⟨*proof*⟩
lemma *compress-normalize-besteffort-not-introduces-Prot-negated*:
¬ *has-disc-negated is-Prot False m* $\Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$ *compress-normalize-besteffort m* = *Some m'* $\Longrightarrow$

¬ *has-disc-negated is-Prot False m'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*

⟨*proof*⟩

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'*
⟨*proof*⟩
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*⟩

lemma *remdups-rev-simplers*: *simple-ruleset rs* ⟹ *simple-ruleset* (*remdups-rev rs*)
⟨*proof*⟩

lemma *remdups-rev-preserve-matches*:
∀ *r*∈*set rs*. *P* (*get-match r*) ⟹ ∀ *r*∈*set* (*remdups-rev rs*). *P* (*get-match r*)
⟨*proof*⟩

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** *wfα*: *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*⟩

theorem *transform-optimize-dnf-strict*:
assumes *simplers*: *simple-ruleset rs* **and** *wfα*: *wf-unknown-match-tac α*
shows (*common-matcher*, *α*),*p*⊢ ⟨*transform-optimize-dnf-strict rs*, *s*⟩ $\Rightarrow_\alpha$ *t* ⟷ (*common-matcher*, *α*),*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *t*
⟨*proof*⟩

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 *wfα*: *wf-unknown-match-tac α* **and** *packet-independent-α*: *packet-independent-α α*
and *wfβ*: *packet-independent-β-unknown β*
shows (*β*, *α*),*p*⊢ ⟨*transform-remove-unknowns-generic* (*β*, *α*) *rs*, *s*⟩ $\Rightarrow_\alpha$ *t* ⟷ (*β*, *α*),*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *t*
and *simple-ruleset* (*transform-remove-unknowns-generic* (*β*, *α*) *rs*)
and ∀ *r* ∈ *set rs*. ¬ *has-disc disc* (*get-match r*) ⟹
∀ *r* ∈ *set* (*transform-remove-unknowns-generic* (*β*, *α*) *rs*). ¬ *has-disc disc* (*get-match r*)
and ∀ *r* ∈ *set* (*transform-remove-unknowns-generic* (*β*, *α*) *rs*). ¬ *has-unknowns β* (*get-match r*)

and ∀ *r* ∈ *set rs*. *normalized-n-primitive disc-sel f* (*get-match r*) ⟹

∀ *r* ∈ *set* (*transform-remove-unknowns-generic* (*β*, *α*) *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-remove-unknowns-generic* (*β*, *α*) *rs*). ¬ *has-disc-negated disc neg* (*get-match r*)
⟨*proof*⟩
thm *transform-remove-unknowns-generic*[*OF - - - packet-independent-β-unknown-common-matcher*]

corollary *transform-remove-unknowns-upper*: **defines** *upper* ≡ *optimize-matches-a upper-closure-matchexpr*
assumes *simplers*: *simple-ruleset rs*
shows (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*upper rs*, *s*⟩ $\Rightarrow_\alpha$ *t* ⟷ (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *t*
and *simple-ruleset* (*upper rs*)
and ∀ *r* ∈ *set rs*. ¬ *has-disc disc* (*get-match r*) ⟹
∀ *r* ∈ *set* (*upper rs*). ¬ *has-disc disc* (*get-match r*)
and ∀ *r* ∈ *set* (*upper rs*). ¬ *has-disc is-Extra* (*get-match r*)
and ∀ *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*)
and ∀ *r* ∈ *set rs*. ¬ *has-disc-negated disc neg* (*get-match r*) ⟹
∀ *r* ∈ *set* (*upper rs*). ¬ *has-disc-negated disc neg* (*get-match r*)
⟨*proof*⟩

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*⟩ $\Rightarrow_\alpha$ *t* ⟷ (*common-matcher*, *in-doubt-deny*),*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *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*⟩

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*=*λm*. *normalized-nnf-match m* ∧ *normalized-n-primitive* (*disc2*, *sel2*) *P m*
and *f*=*normalize-primitive-extract* (*disc1*, *sel1*) *C f*]
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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 *wf* α: *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*⟩ $\Rightarrow_\alpha$ *t* ⟷ (*common-matcher*, α),*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *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*⟩

theorem *iiface-constrain*:

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 *nospoofing*: $\bigwedge$*ips. ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* $\Longrightarrow$ *p-src p* $\in$ *ipcidr-union-set* (*set ips*)
shows (*common-matcher*, α),*p*$\vdash$ $\langle$*optimize-matches* (*iiface-constrain 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-constrain ipassmt*) *rs*)
$\langle$*proof*$\rangle$

In contrast to $[\![$*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*)$]\!]$ $\Longrightarrow$ (*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*
$[\![$*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*)$]\!]$ $\Longrightarrow$ *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*)
$\langle$*proof*$\rangle$

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*$\vdash$ $\langle$*optimize-matches* (*oiface-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* (*oiface-rewrite ipassmt*) *rs*)
$\langle$*proof*$\rangle$

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*⟩ $\Rightarrow_\alpha$ *t* ⟷ (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *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*⟩

lemma *transform-lower-closure*:
assumes *simplers*: *simple-ruleset rs*
— semantics are preserved
shows (*common-matcher*, *in-doubt-deny*),*p*⊢ ⟨*lower-closure rs*, *s*⟩ $\Rightarrow_\alpha$ *t* ⟷ (*common-matcher*, *in-doubt-deny*),*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *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*⟩

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*)
⟨*proof*⟩

lemma *iiface-rewrite-preserves-nodisc*:
∀ *a*. ¬ *disc* (*Src a*) ⟹ ¬ *has-disc disc m* ⟹ ¬ *has-disc disc* (*iiface-rewrite ipassmt m*)
⟨*proof*⟩

lemma *iiface-constrain-preserves-nodisc*:
∀ *a*. ¬ *disc* (*Src a*) ⟹ ¬ *has-disc disc m* ⟹ ¬ *has-disc disc* (*iiface-constrain ipassmt m*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩ $\Rightarrow_\alpha$ *t* ⟷ (*common-matcher*, α),*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *t*
⟨*proof*⟩

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*)
⟨*proof*⟩

context begin

private lemma *iiface-rewrite-monoNone*: *matcheq-matchNone m* $\Longrightarrow$ *matcheq-matchNone* (*iiface-rewrite ipassmt m*)
⟨*proof*⟩ **lemma** *iiface-constrain-monoNone*: *matcheq-matchNone m* $\Longrightarrow$ *matcheq-matchNone* (*iiface-constrain ipassmt m*)
⟨*proof*⟩ **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*: $\forall$ *r* $\in$ *set rs*. *normalized-nnf-match* (*get-match r*)
and *wf-ipassmt*: *ipassmt-sanity-nowildcards* (*map-of ipassmt*) *distinct* (*map fst ipassmt*)
and *nospoofing*: $\exists$ *ips*. (*map-of ipassmt*) (*Iface* (*p-iiface p*)) = *Some ips* $\wedge$ *p-src p* $\in$ *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*$\vdash$ ⟨*iface-try-rewrite ipassmt* (*Some rtbl*) *rs*, *s*⟩ $\Rightarrow_\alpha$ *t* $\longleftrightarrow$ (*common-matcher*, α),*p*$\vdash$ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *t*
⟨*proof*⟩

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* $\Rightarrow$ *'i::len common-primitive match-expr* $\Rightarrow$ *'i common-primitive match-expr* **where**
ctstate-assume-state s (*Match* (*CT-State x*)) = (*if s* $\in$ *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* $\Longrightarrow$
matches (*common-matcher*, α) (*ctstate-assume-state s m*) *a p* $\longleftrightarrow$ *matches* (*common-matcher*, α) *m a p*

⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

corollary *ipt-tcp-flags-assume-syn*: *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ⟹

approximating-bigstep-fun (*common-matcher*, α) *p* (*ipt-tcp-flags-assume-syn rs*) *s* = *approximating-bigstep-fun* (*common-matcher*, α) *p rs s*
⟨*proof*⟩

definition *packet-assume-new* :: *'i::len common-primitive rule list* ⇒ *'i common-primitive rule list* **where**
packet-assume-new ≡ *ctstate-assume-new* ∘ *ipt-tcp-flags-assume-syn*

lemma *packet-assume-new-simple-ruleset*: *simple-ruleset rs* ⟹ *simple-ruleset* (*packet-assume-new rs*)
⟨*proof*⟩

corollary *packet-assume-new*: *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) ⟹ *p-tag-ctstate p* = *CT-New* ⟹
approximating-bigstep-fun (*common-matcher*, α) *p* (*packet-assume-new rs*) *s* = *approximating-bigstep-fun* (*common-matcher*, α) *p rs s*
⟨*proof*⟩

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* ⇒ *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* (*λr. 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*)
⟨*proof*⟩
lemma *abstract-primitive-preserves-nodisc*:
¬ *has-disc disc' m* ⟹ (∀ *str*. ¬ *disc'* (*Extra str*)) ⟹ ¬ *has-disc disc'* (*abstract-primitive disc m*)
⟨*proof*⟩
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*)
⟨*proof*⟩

lemma *abstract-primitive-nodisc*:
∀ *x*. *disc' x* ⟶ *disc* (*Pos x*) ∧ *disc* (*Neg x*) ⟹ (∀ *str*. ¬ *disc'* (*Extra str*)) ⟹ ¬ *has-disc disc'* (*abstract-primitive disc m*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

context
begin
private lemma *abstract-primitive-in-doubt-allow-Allow*:
primitive-matcher-generic β ⟹ *normalized-nnf-match m* ⟹
matches (β, *in-doubt-allow*) *m action.Accept p* ⟹
matches (β, *in-doubt-allow*) (*abstract-primitive disc m*) *action.Accept p*
⟨*proof*⟩ **lemma** *abstract-primitive-in-doubt-allow-Allow2*:
primitive-matcher-generic β ⟹ *normalized-nnf-match m* ⟹
¬ *matches* (β, *in-doubt-allow*) *m action.Drop p* ⟹
¬ *matches* (β, *in-doubt-allow*) (*abstract-primitive disc m*) *action.Drop p*
⟨*proof*⟩ **lemma** *abstract-primitive-in-doubt-allow-Deny*:
primitive-matcher-generic β ⟹ *normalized-nnf-match m* ⟹
matches (β, *in-doubt-allow*) (*abstract-primitive disc m*) *action.Drop p* ⟹
matches (β, *in-doubt-allow*) *m action.Drop p*
⟨*proof*⟩ **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*
⟨*proof*⟩

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*⊢ ⟨*abstract disc rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny*} $\subseteq$ {*p*. γ,*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny*}
(**is** *?deny*)
and {*p*. γ,*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow*} $\subseteq$ {*p*. γ,*p*⊢ ⟨*abstract disc rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow*}
(**is** *?allow*)
⟨*proof*⟩
corollary *abstract-primitive-in-doubt-allow*:
assumes $\forall$ $r \in$ *set rs*. *normalized-nnf-match* (*get-match r*) **and** *simple-ruleset rs*
defines $\gamma \equiv$ (*common-matcher*, *in-doubt-allow*) **and** *abstract disc* $\equiv$ *optimize-matches* (*abstract-primitive disc*)
shows {*p*. γ,*p*⊢ ⟨*abstract disc rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny*} $\subseteq$ {*p*. γ,*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny*}
and {*p*. γ,*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow*} $\subseteq$ {*p*. γ,*p*⊢ ⟨*abstract disc rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow*}
⟨*proof*⟩
end

context
begin
private lemma *abstract-primitive-in-doubt-deny-Deny*:
primitive-matcher-generic $\beta \Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$
matches (β, *in-doubt-deny*) *m action.Drop p* $\Longrightarrow$
matches (β, *in-doubt-deny*) (*abstract-primitive disc m*) *action.Drop p*
⟨*proof*⟩ **lemma** *abstract-primitive-in-doubt-deny-Deny2*:
primitive-matcher-generic $\beta \Longrightarrow$ *normalized-nnf-match m* $\Longrightarrow$
$\neg$ *matches* (β, *in-doubt-deny*) *m action.Accept p* $\Longrightarrow$
$\neg$ *matches* (β, *in-doubt-deny*) (*abstract-primitive disc m*) *action.Accept p*
⟨*proof*⟩ **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*
⟨*proof*⟩ **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* $\Longrightarrow$

¬ *matches* (β, *in-doubt-deny*) *m action.Drop p*
⟨*proof*⟩

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*: ∀ *r* ∈ *set rs*. *normalized-nnf-match* (*get-match r*)
and *simple*: *simple-ruleset rs*
defines γ ≡ (β, *in-doubt-deny*) **and** *abstract disc* ≡ *optimize-matches* (*abstract-primitive disc*)
shows {*p*. γ,*p*⊢ ⟨*abstract disc rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow*} ⊆ {*p*. γ,*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow*}
(**is** *?allow*)
and {*p*. γ,*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny*} ⊆ {*p*. γ,*p*⊢ ⟨*abstract disc rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny*}
(**is** *?deny*)
⟨*proof*⟩
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
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *simple-fw-remdups-Rev*: *simple-fw* (*remdups-rev rs*) *p* = *simple-fw rs p*
⟨*proof*⟩

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*
⟨*proof*⟩
lemma ¬ *matcheq-matchNone m* ⟹ *normalized-src-ports m* ⟹ *normalized-nnf-match m*
⟨*proof*⟩

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 (*λr. 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*
⟨*proof*⟩

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] ⟨*proof*⟩
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] ⟨*proof*⟩
lemma *check-simple-fw-preconditions* [*Rule* (*MatchAnd MatchAny MatchAny*) *Drop*]
⟨*proof*⟩

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] ⟨*proof*⟩
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] ⟨*proof*⟩

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*⟩

lemma *ctstate-assume-new-not-has-CT-State*:
r ∈ *set* (*ctstate-assume-new rs*) ⟹ ¬ *has-disc is-CT-State* (*get-match r*)
⟨*proof*⟩

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*⟩ $\Rightarrow_\alpha$ *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
⟨*proof*⟩

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*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow* ∧ *newpkt p*}
⟨*proof*⟩

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*: $\bigwedge$*rtbl. rtblo* = *Some rtbl* $\Longrightarrow$ *correct-routing rtbl*
— A passed routing table contains no interfaces with wildcard names
and *routing-no-wildcards*: $\bigwedge$*rtbl. rtblo* = *Some rtbl* $\Longrightarrow$ *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*⟩ $\Rightarrow_\alpha$ *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*⟩

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*: Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided*
shows (β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Undecided* ∨ (β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow*
⟨*proof*⟩

lemma *FinalAllow-approximating-in-doubt-allow*:
assumes *agree*: *matcher-agree-on-exact-matches* γ β
and *good*: *good-ruleset rs* **and** *semantics*: Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*
shows (β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow*
⟨*proof*⟩

corollary *FinalAllows-subseteq-in-doubt-allow*: *matcher-agree-on-exact-matches* γ β $\Longrightarrow$ *good-ruleset rs* $\Longrightarrow$
{*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*} ⊆ {*p*. (β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow*}
⟨*proof*⟩

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*⟩

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*⟩ $\Rightarrow_\alpha$ *Undecided* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided* ∨ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalDeny*
⟨*proof*⟩

lemma *FinalDeny-approximating-in-doubt-allow*: *matcher-agree-on-exact-matches* γ β ⟹
good-ruleset rs ⟹
(β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalDeny*
⟨*proof*⟩

corollary *FinalDenys-subseteq-in-doubt-allow*: *matcher-agree-on-exact-matches* γ β ⟹ *good-ruleset rs* ⟹
{*p*. (β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny*} ⊆ {*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalDeny*}
⟨*proof*⟩

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 ⟹ *good-ruleset rs* ⟹
approximating-bigstep-fun (β, *in-doubt-allow*) *p rs Undecided* = (*Decision FinalDeny*) ⟹ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalDeny*
⟨*proof*⟩

42.2 Tactic *in-doubt-deny*

lemma *iptables-bigstep-undecided-to-undecided-in-doubt-deny-approx*: *matcher-agree-on-exact-matches* γ β ⟹

good-ruleset rs ⟹
Γ,γ,p⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided* ⟹
(β, *in-doubt-deny*),p⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Undecided* ∨ (β, *in-doubt-deny*),p⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny*
⟨*proof*⟩

lemma *FinalDeny-approximating-in-doubt-deny*: *matcher-agree-on-exact-matches* γ β ⟹
good-ruleset rs ⟹
Γ,γ,p⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalDeny* ⟹ (β, *in-doubt-deny*),p⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny*
⟨*proof*⟩

lemma *approximating-bigstep-undecided-to-undecided-in-doubt-deny-approx*: *matcher-agree-on-exact-matches* γ β ⟹
good-ruleset rs ⟹
(β, *in-doubt-deny*),p⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Undecided* ⟹ Γ,γ,p⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided* ∨ Γ,γ,p⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*
⟨*proof*⟩

lemma *FinalAllow-approximating-in-doubt-deny*: *matcher-agree-on-exact-matches* γ β ⟹
good-ruleset rs ⟹
(β, *in-doubt-deny*),p⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow* ⟹ Γ,γ,p⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*
⟨*proof*⟩

corollary *FinalAllows-subseteq-in-doubt-deny*: *matcher-agree-on-exact-matches* γ β ⟹ *good-ruleset rs* ⟹
{p. (β, *in-doubt-deny*),p⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow*} ⊆ {p. Γ,γ,p⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*}
⟨*proof*⟩

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*

$\wedge$ *newpkt p*} $\subseteq$ {*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow* ∧ *newpkt p*}
⟨*proof*⟩

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*⟩ $\Rightarrow_\alpha$ *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*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow*}
⟨*proof*⟩

theorem *FinalDenyClosure*:
assumes *matcher-agree-on-exact-matches* γ β **and** *good-ruleset rs*
shows {*p*. (β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *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*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny*}
⟨*proof*⟩

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*⟩ $\Rightarrow_\alpha$ *t* ⟷ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t*
⟨*proof*⟩

For rulesets without *Call*s, the approximating ternary semantics can perfectly simulate the Boolean semantics.

theorem β_{magic}*-approximating-bigstep-iff-iptables-bigstep*:
assumes ∀ *r* ∈ *set rs*. ∀ *c*. *get-action r* ≠ *Call c*
shows ((β_{magic} γ),α),*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *t* ⟷ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t*
⟨*proof*⟩

corollary β_{magic}*-approximating-bigstep-fun-iff-iptables-bigstep*:
assumes *good-ruleset rs*
shows *approximating-bigstep-fun* (β_{magic} γ,α) *p rs s* = *t* ⟷ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *t*
⟨*proof*⟩

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*⟩

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 $\Gamma,\gamma,p\vdash$ ⟨*normalize-rules-dnf rs, s*⟩ $\Rightarrow$ *t* $\longleftrightarrow$ $\Gamma,\gamma,p\vdash$ ⟨*rs, s*⟩ $\Rightarrow$ *t*
⟨*proof*⟩

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* $\Rightarrow$ *string* **where**
quote-rewrite $\equiv$ *map* (λ*c. if c = char-of-nat 34 then CHR ''~'' else c*)

lemma *quote-rewrite* (*''foo''*@[*char-of-nat 34*]) = *''foo~''* ⟨*proof*⟩

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* $\times$ *32 common-primitive rule list*) *list* $\Rightarrow$ *string* $\rightharpoonup$ *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'*
⟨*proof*⟩

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*] ⟨*proof*⟩

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''] ⟨*proof*⟩ **lemma** *eval-ternary-And-Unknown-Unkown*:
eval-ternary-And TernaryUnknown (*eval-ternary-And TernaryUnknown tv*) =
eval-ternary-And TernaryUnknown tv
⟨*proof*⟩ **lemma** *is-pos-Extra-alist-and*:
is-pos-Extra a ⟹ *alist-and* (*a#as*) = *MatchAnd* (*Match* (*Extra* (*get-pos-Extra a*))) (*alist-and as*)
⟨*proof*⟩ **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*⟩

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*)
⟨*proof*⟩
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*)]))] ⟨*proof*⟩

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*'')] ⟨*proof*⟩
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*) ⟨*proof*⟩
lemma *prefix-to-strange-inverse-cisco-mask 16* = (*0*, *0*, *255*, *255*) ⟨*proof*⟩
lemma *prefix-to-strange-inverse-cisco-mask 24* = (*0*, *0*, *0*, *255*) ⟨*proof*⟩
lemma *prefix-to-strange-inverse-cisco-mask 32* = (*0*, *0*, *0*, *0*) ⟨*proof*⟩

end

45 Parser for iptables-save

theory *Parser6*
imports *Code-Interface*
keywords *parse-ip6tables-save* :: *thy-decl*
begin

⟨*ML*⟩

46 An SML Parser for iptables-save

Work in Progress

⟨*ML*⟩

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*|)$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow*) $\longrightarrow$
p-src p $\in$ (*ipcidr-union-set* (*set* (*the* (*ipassmt iface*))))
$\langle$*proof*$\rangle$

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* $\in$ *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*)))
$\langle$*proof*$\rangle$

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*⟩

corollary *no-spoofing-executable-set-preprocessed*:
fixes *ipassmt* :: *'i::len ipassignment*
defines *preprocess rs* ≡ *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 *matcher-agree-on-exact-matches* γ *common-matcher*
and *simplers*: *simple-ruleset rs*
and *no-spoofing-executable*: ∀ *iface* ∈ *dom ipassmt*. *no-spoofing-iface iface ipassmt* (*preprocess rs*)
and *iface* ∈ *dom ipassmt*
shows {*p-src p* | *p* :: (*'i::len,'pkt-ext*) *tagged-packet-scheme*. *newpkt p* ∧ Γ,γ,*p*(|*p-iiface*:=*iface-sel iface*|)⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*} ⊆
ipcidr-union-set (*set* (*the* (*ipassmt iface*)))
⟨*proof*⟩

end

48 Parser for iptables-save

theory *Parser*
imports *Code-Interface*
keywords *parse-iptables-save* :: *thy-decl*
begin

⟨*ML*⟩

49 An SML Parser for iptables-save

Work in Progress

⟨*ML*⟩

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*⟩

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*: $\Gamma,\gamma,p\vdash$ ⟨*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*⟩

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:

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

$$\frac{\textit{Semantics-Goto.matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle [\textit{Rule}\ m\ \textit{action.Accept}],\ \textit{Undecided}\rangle \Rightarrow \textit{Decision FinalAllow}}$$

$$\frac{\textit{Semantics-Goto.matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle [\textit{Rule}\ m\ \textit{action.Drop}],\ \textit{Undecided}\rangle \Rightarrow \textit{Decision FinalDeny}}$$

$$\frac{\textit{Semantics-Goto.matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle [\textit{Rule}\ m\ \textit{Reject}],\ \textit{Undecided}\rangle \Rightarrow \textit{Decision FinalDeny}}$$

$$\frac{\textit{Semantics-Goto.matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle [\textit{Rule}\ m\ \textit{Log}],\ \textit{Undecided}\rangle \Rightarrow \textit{Undecided}}$$

$$\frac{\textit{Semantics-Goto.matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[\textit{Rule m Empty}],\ \textit{Undecided}\rangle \Rightarrow \textit{Undecided}}$$

$$\frac{\neg\ \textit{Semantics-Goto.matches}\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash_g \langle[\textit{Rule m a}],\ \textit{Undecided}\rangle \Rightarrow \textit{Undecided}}$$

$$\Gamma,\gamma,p\vdash_g \langle rs,\ \textit{Decision X}\rangle \Rightarrow \textit{Decision X}$$

$$\frac{\Gamma,\gamma,p\vdash_g \langle rs_1,\ \textit{Undecided}\rangle \Rightarrow t \quad \Gamma,\gamma,p\vdash_g \langle rs_2,\ t\rangle \Rightarrow t' \qquad \textit{Semantics-Goto.no-matching-Goto}\ \gamma\ p\ rs_1}{\Gamma,\gamma,p\vdash_g \langle rs_1\ @\ rs_2,\ \textit{Undecided}\rangle \Rightarrow t'}$$

$$\frac{\begin{array}{c}\textit{Semantics-Goto.matches}\ \gamma\ m\ p \\ \Gamma\ \textit{chain} = \textit{Some}\ (rs_1\ @\ [\textit{Rule}\ m'\ \textit{Return}]\ @\ rs_2) \\ \textit{Semantics-Goto.matches}\ \gamma\ m'\ p \\ \Gamma,\gamma,p\vdash_g \langle rs_1,\ \textit{Undecided}\rangle \Rightarrow \textit{Undecided} \\ \textit{Semantics-Goto.no-matching-Goto}\ \gamma\ p\ rs_1\end{array}}{\Gamma,\gamma,p\vdash_g \langle[\textit{Rule m (Call chain)}],\ \textit{Undecided}\rangle \Rightarrow \textit{Undecided}}$$

$$\frac{\textit{Semantics-Goto.matches}\ \gamma\ m\ p \quad \Gamma\ \textit{chain} = \textit{Some rs} \qquad \Gamma,\gamma,p\vdash_g \langle rs,\ \textit{Undecided}\rangle \Rightarrow t}{\Gamma,\gamma,p\vdash_g \langle[\textit{Rule m (Call chain)}],\ \textit{Undecided}\rangle \Rightarrow t}$$

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'* $\Longrightarrow \Gamma,\gamma,p\vdash_g \langle rs', s\rangle \Rightarrow t = \Gamma,\gamma,p\vdash_g \langle rs, s\rangle \Rightarrow t$

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* ⇒ (*'i*, *'pkt-ext*) *tagged-packet-scheme* ⇒ *bool*
and *p* :: (*'i::len*, *'pkt-ext*) *tagged-packet-scheme*
assumes *sanity-wf-ruleset* Γ **and** *chain-name* ∈ *set* (*map fst* Γ)
and *default-action* = *action.Accept* ∨ *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*⊢ ⟨*rs*, *s*⟩ ⇒ *t* ⟷
(*map-of* Γ),γ,*p*⊢ ⟨[*Rule MatchAny* (*Call chain-name*), *Rule MatchAny default-action*], *s*⟩ ⇒ *t*
and *simple-ruleset rs*
⟨*proof*⟩

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*; ∀ *r*∈*set rs*. *normalized-nnf-match* (*get-match r*); ∀ *iface*∈*dom ipassmt*. *no-spoofing-iface iface ipassmt rs*; *iface* ∈ *dom ipassmt*⟧ ⟹ {*p-src p* | Γ,γ,*p*⦇*p-iiface* := *iface-sel iface*⦈⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Decision FinalAllow*} ⊆ *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*

$(\mathit{lower\text{-}closure}\ (\mathit{packet\text{-}assume\text{-}new}\ \mathit{rs})))))\ p = \mathit{Decision}\ \mathit{FinalAllow} \wedge \mathit{match\text{-}tcp\text{-}flags}\ \mathit{ipt\text{-}tcp\text{-}syn}\ (\mathit{p\text{-}tcp\text{-}flags}\ p) \wedge \mathit{p\text{-}tag\text{-}ctstate}\ p = \mathit{CT\text{-}New}\} \subseteq \{p \mid \Gamma,\gamma,p\vdash \langle \mathit{rs}, \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalAllow} \wedge \mathit{match\text{-}tcp\text{-}flags}\ \mathit{ipt\text{-}tcp\text{-}syn}\ (\mathit{p\text{-}tcp\text{-}flags}\ p) \wedge \mathit{p\text{-}tag\text{-}ctstate}\ p = \mathit{CT\text{-}New}\}$

We support translating to a more permissive version (a version that accepts more packets):

$[\![\mathit{matcher\text{-}agree\text{-}on\text{-}exact\text{-}matches}\ \gamma\ \mathit{common\text{-}matcher};\ \mathit{simple\text{-}ruleset}\ \mathit{rs}]\!] \Longrightarrow \{p \mid \Gamma,\gamma,p\vdash \langle \mathit{rs}, \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalAllow} \wedge \mathit{match\text{-}tcp\text{-}flags}\ \mathit{ipt\text{-}tcp\text{-}syn}\ (\mathit{p\text{-}tcp\text{-}flags}\ p) \wedge \mathit{p\text{-}tag\text{-}ctstate}\ p = \mathit{CT\text{-}New}\} \subseteq \{p \mid \mathit{simple\text{-}fw}\ (\mathit{to\text{-}simple\text{-}firewall}\ (\mathit{upper\text{-}closure}\ (\mathit{optimize\text{-}matches}\ \mathit{abstract\text{-}for\text{-}simple\text{-}firewall}\ (\mathit{upper\text{-}closure}\ (\mathit{packet\text{-}assume\text{-}new}\ \mathit{rs})))))\ p = \mathit{Decision}\ \mathit{FinalAllow} \wedge \mathit{match\text{-}tcp\text{-}flags}\ \mathit{ipt\text{-}tcp\text{-}syn}\ (\mathit{p\text{-}tcp\text{-}flags}\ p) \wedge \mathit{p\text{-}tag\text{-}ctstate}\ p = \mathit{CT\text{-}New}\}$

There is also a different approach to translate to the simple firewall which removes all matches on interfaces:

$[\![\mathit{simple\text{-}ruleset}\ \mathit{rs};\ \mathit{ipassmt\text{-}sanity\text{-}nowildcards}\ (\mathit{map\text{-}of}\ \mathit{ipassmt});\ \mathit{distinct}\ (\mathit{map}\ \mathit{fst}\ \mathit{ipassmt});\ \forall p.\ \exists \mathit{ips}.\ \mathit{map\text{-}of}\ \mathit{ipassmt}\ (\mathit{Iface}\ (\mathit{p\text{-}iiface}\ p)) = \mathit{Some}\ \mathit{ips} \wedge \mathit{p\text{-}src}\ p \in \mathit{ipcidr\text{-}union\text{-}set}\ (\mathit{set}\ \mathit{ips});\ \bigwedge \mathit{rtbl}\ p.\ \mathit{rtblo} = \mathit{Some}\ \mathit{rtbl} \Longrightarrow \mathit{output\text{-}iface}\ (\mathit{routing\text{-}table\text{-}semantics}\ \mathit{rtbl}\ (\mathit{p\text{-}dst}\ p)) = \mathit{p\text{-}oiface}\ p;\ \bigwedge \mathit{rtbl}.\ \mathit{rtblo} = \mathit{Some}\ \mathit{rtbl} \Longrightarrow \mathit{correct\text{-}routing}\ \mathit{rtbl};\ \bigwedge \mathit{rtbl}.\ \mathit{rtblo} = \mathit{Some}\ \mathit{rtbl} \Longrightarrow \mathit{ipassmt\text{-}sanity\text{-}nowildcards}\ (\mathit{map\text{-}of}\ (\mathit{routing\text{-}ipassmt}\ \mathit{rtbl}))]\!] \Longrightarrow \{p \mid (\mathit{common\text{-}matcher}, \mathit{in\text{-}doubt\text{-}allow}),p\vdash \langle \mathit{rs}, \mathit{Undecided}\rangle \Rightarrow_\alpha \mathit{Decision}\ \mathit{FinalAllow} \wedge \mathit{match\text{-}tcp\text{-}flags}\ \mathit{ipt\text{-}tcp\text{-}syn}\ (\mathit{p\text{-}tcp\text{-}flags}\ p) \wedge \mathit{p\text{-}tag\text{-}ctstate}\ p = \mathit{CT\text{-}New}\} \subseteq \{p \mid \mathit{simple\text{-}fw}\ (\mathit{to\text{-}simple\text{-}firewall\text{-}without\text{-}interfaces}\ \mathit{ipassmt}\ \mathit{rtblo}\ \mathit{rs})\ p = \mathit{Decision}\ \mathit{FinalAllow} \wedge \mathit{match\text{-}tcp\text{-}flags}\ \mathit{ipt\text{-}tcp\text{-}syn}\ (\mathit{p\text{-}tcp\text{-}flags}\ p) \wedge \mathit{p\text{-}tag\text{-}ctstate}\ p = \mathit{CT\text{-}New}\}$

$[\![\mathit{simple\text{-}ruleset}\ \mathit{rs};\ \mathit{ipassmt\text{-}sanity\text{-}nowildcards}\ (\mathit{map\text{-}of}\ \mathit{ipassmt});\ \mathit{distinct}\ (\mathit{map}\ \mathit{fst}\ \mathit{ipassmt});\ \forall p.\ \exists \mathit{ips}.\ \mathit{map\text{-}of}\ \mathit{ipassmt}\ (\mathit{Iface}\ (\mathit{p\text{-}iiface}\ p)) = \mathit{Some}\ \mathit{ips} \wedge \mathit{p\text{-}src}\ p \in \mathit{ipcidr\text{-}union\text{-}set}\ (\mathit{set}\ \mathit{ips});\ \bigwedge \mathit{rtbl}\ p.\ \mathit{rtblo} = \mathit{Some}\ \mathit{rtbl} \Longrightarrow \mathit{output\text{-}iface}\ (\mathit{routing\text{-}table\text{-}semantics}\ \mathit{rtbl}\ (\mathit{p\text{-}dst}\ p)) = \mathit{p\text{-}oiface}\ p;\ \bigwedge \mathit{rtbl}.\ \mathit{rtblo} = \mathit{Some}\ \mathit{rtbl} \Longrightarrow \mathit{correct\text{-}routing}\ \mathit{rtbl};\ \bigwedge \mathit{rtbl}.\ \mathit{rtblo} = \mathit{Some}\ \mathit{rtbl} \Longrightarrow \mathit{ipassmt\text{-}sanity\text{-}nowildcards}\ (\mathit{map\text{-}of}\ (\mathit{routing\text{-}ipassmt}\ \mathit{rtbl}))]\!] \Longrightarrow \forall r\in \mathit{set}\ (\mathit{to\text{-}simple\text{-}firewall\text{-}without\text{-}interfaces}\ \mathit{ipassmt}\ \mathit{rtblo}\ \mathit{rs}).\ \mathit{iiface}\ (\mathit{match\text{-}sel}\ r) = \mathit{ifaceAny} \wedge \mathit{oiface}\ (\mathit{match\text{-}sel}\ r) = \mathit{ifaceAny}$

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: $V \in \mathit{set}\ (\mathit{build\text{-}ip\text{-}partition}\ c\ \mathit{rs}) \Longrightarrow \forall \mathit{ip1}\in \mathit{wordinterval\text{-}to\text{-}set}\ V.\ \forall \mathit{ip2}\in \mathit{wordinterval\text{-}to\text{-}set}\ V.\ \mathit{same\text{-}fw\text{-}behaviour\text{-}one}\ \mathit{ip1}\ \mathit{ip2}\ c\ \mathit{rs}$

Minimal: $[\![A \in \mathit{set}\ (\mathit{build\text{-}ip\text{-}partition}\ c\ \mathit{rs});\ B \in \mathit{set}\ (\mathit{build\text{-}ip\text{-}partition}\ c\ \mathit{rs});\ A \neq B]\!] \Longrightarrow \forall \mathit{ip1}\in \mathit{wordinterval\text{-}to\text{-}set}\ A.\ \forall \mathit{ip2}\in \mathit{wordinterval\text{-}to\text{-}set}\ B.$

$\neg$ *same-fw-behaviour-one ip1 ip2 c rs*

The resulting access control matrix is sound and complete:

(V, E) = *access-matrix c rs* $\Longrightarrow$ ($\exists$ *s-repr d-repr s-range d-range*. (*s-repr*, *d-repr*) $\in$ *set E* $\wedge$ *map-of V s-repr* = *Some s-range* $\wedge$ *s* $\in$ *wordinterval-to-set s-range* $\wedge$ *map-of V d-repr* = *Some d-range* $\wedge$ *d* $\in$ *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*:

$\llbracket$*matcher-agree-on-exact-matches* γ *common-matcher*; *simple-ruleset rs*; *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) $\wedge$ *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-sport p*, *pc-dport* = *p-dport p*|) (*to-simple-firewall* (*upper-closure* (*optimize-matches abstract-for-simple-firewall* (*upper-closure* (*packet-assume-new rs*))))); $\Gamma,\gamma,p\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow*$\rrbracket$ $\Longrightarrow$ $\exists$ *s-repr d-repr s-range d-range*. (*s-repr*, *d-repr*) $\in$ *set E* $\wedge$ *map-of V s-repr* = *Some s-range* $\wedge$ *p-src p* $\in$ *wordinterval-to-set s-range* $\wedge$ *map-of V d-repr* = *Some d-range* $\wedge$ *p-dst p* $\in$ *wordinterval-to-set d-range*

Actually, we want to ignore all interfaces for a service matrix. This is done in $\llbracket$*matcher-agree-on-exact-matches* γ *common-matcher*; *simple-ruleset rs*; *ipassmt-sanity-nowildcards* (*map-of ipassmt*); *distinct* (*map fst ipassmt*); $\forall$ *p*. $\exists$ *ips*. *map-of ipassmt* (*Iface* (*p-iiface p*)) = *Some ips* $\wedge$ *p-src p* $\in$ *ipcidr-union-set* (*set ips*); $\bigwedge$*rtbl p*. *rtblo* = *Some rtbl* $\Longrightarrow$ *output-iface* (*routing-table-semantics rtbl* (*p-dst p*)) = *p-oiface p*; $\bigwedge$*rtbl*. *rtblo* = *Some rtbl* $\Longrightarrow$ *correct-routing rtbl*; $\bigwedge$*rtbl*. *rtblo* = *Some rtbl* $\Longrightarrow$ *ipassmt-sanity-nowildcards* (*map-of* (*routing-ipassmt rtbl*)); *match-tcp-flags ipt-tcp-syn* (*p-tcp-flags p*) $\wedge$ *p-tag-ctstate p* = *CT-New*; (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*); $\Gamma,\gamma,p\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow*$\rrbracket$ $\Longrightarrow$ $\exists$ *s-repr d-repr s-range d-range*. (*s-repr*, *d-repr*) $\in$ *set E* $\wedge$ *map-of V s-repr* = *Some s-range* $\wedge$ *p-src p* $\in$ *wordinterval-to-set s-range* $\wedge$ *map-of V d-repr* = *Some d-range* $\wedge$ *p-dst p* $\in$ *wordinterval-to-set d-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.