

The Unified Policy Framework (UPF)

Achim D. Brucker* Lukas Brügger[†] Burkhard Wolff[‡]

*SAP SE, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany

achim.brucker@sap.com

Information Security, ETH Zurich, 8092 Zurich, Switzerland

Lukas.A.Bruegger@gmail.com

[‡]Univ. Paris-Sud, Laboratoire LRI, UMR8623, 91405 Orsay, France France

burkhard.wolff@lri.fr

February 6, 2026

Abstract

We present the *Unified Policy Framework* (UPF), a generic framework for modelling security (access-control) policies; in Isabelle/HOL. UPF emphasizes the view that a policy is a policy decision function that grants or denies access to resources, permissions, etc. In other words, instead of modelling the relations of permitted or prohibited requests directly, we model the concrete function that implements the policy decision point in a system, seen as an “aspect” of “wrapper” around the business logic of a system. In more detail, UPF is based on the following four principles: 1. Functional representation of policies, 2. No conflicts are possible, 3. Three-valued decision type (allow, deny, undefined), 4. Output type not containing the decision only.

Contents

1	Introduction	5
2	The Unified Policy Framework (UPF)	7
2.1	The Core of the Unified Policy Framework (UPF)	7
2.1.1	Foundation	7
2.1.2	Policy Constructors	8
2.1.3	Override Operators	9
2.1.4	Coercion Operators	11
2.2	Elementary Policies	14
2.2.1	The Core Policy Combinators: Allow and Deny Everything	15
2.2.2	Common Instances	16
2.2.3	Domain, Range, and Restrictions	17
2.3	Sequential Composition	22
2.3.1	Flattening	22
2.3.2	Policy Composition	24
2.4	Parallel Composition	26
2.4.1	Parallel Combinators: Foundations	26
2.4.2	Combinators for Transition Policies	30
2.4.3	Range Splitting	30
2.4.4	Distributivity of the parallel combinators	31
2.5	Properties on Policies	33
2.5.1	Basic Properties	33
2.5.2	Combined Data-Policy Refinement	35
2.5.3	Equivalence of Policies	36
2.6	Policy Transformations	37
2.6.1	Elementary Operators	37
2.6.2	Distributivity of the Transformation.	41
2.7	Policy Transformation for Testing	45
2.8	Putting Everything Together: UPF	48
3	Example	49
3.1	Secure Service Specification	49
3.1.1	Datatypes for Modelling Users and Roles	49
3.1.2	Modelling Health Records and the Web Service API	50
3.1.3	Modelling Access Control	53
3.1.4	The State Transitions and Output Function	57
3.1.5	Combine All Parts	58

3.2	Instantiating Our Secure Service Example	59
3.2.1	Access Control Configuration	59
3.2.2	The Initial System State	60
3.2.3	Basic Properties	60
4	Conclusion and Related Work	63
4.1	Related Work	63
4.2	Conclusion Future Work	63
5	Appendix	65
5.1	Basic Monad Theory for Sequential Computations	65
5.1.1	General Framework for Monad-based Sequence-Test	65
5.1.2	Valid Test Sequences in the State Exception Monad	73
5.1.3	Valid Test Sequences in the State Exception Backtrack Monad	76

1 Introduction

Access control, i. e., restricting the access to information or resources, is an important pillar of today’s information security portfolio. Thus the large number of access control models (e. g., [1, 5, 6, 15–17, 19, 21]) and variants thereof (e. g., [2, 2, 4, 7, 14, 18, 22]) is not surprising. On the one hand, this variety of specialized access control models allows concise representation of access control policies. On the other hand, the lack of a common foundations makes it difficult to compare and analyze different access control models formally.

We present formalization of the Unified Policy Framework (UPF) [13] that provides a formal semantics for the core concepts of access control policies. It can serve as a meta-model for a large set of well-known access control policies and moreover, serve as a framework for analysis and test generation tools addressing common ground in policy models. Thus, UPF for comparing different access control models, including a formal correctness proof of a specific embedding, for example, implementing a role-based access control policy in terms of a discretionary access enforcement architecture. Moreover, defining well-known access control models by instantiating a unified policy framework allows to re-use tools, such as test-case generators, that are already provided for the unified policy framework. As the instantiation of a unified policy framework may also define a domain-specific (i. e., access control model specific) set of policy combinators (syntax), such an approach still provides the usual notations and thus a concise representation of access control policies.

UPF was already successful used as a basis for large scale access control policies in the health care domain [10] as well as in the domain of firewall and router policies [12]. In both domains, the formal policy specifications served as basis for the generation, using HOL-TestGen [9], of test cases that can be used for validating the compliance of an implementation to the formal model. UPF is based on the following four principles:

1. policies are represented as *functions* (rather than relations),
2. policy combination avoids conflicts by construction,
3. the decision type is three-valued (allow, deny, undefined),
4. the output type does not only contain the decision but also a ‘slot’ for arbitrary result data.

UPF is related to the state-exception monad modeling failing computations; in some cases our UPF model makes explicit use of this connection, although it is not central. The used theory for state-exception monads can be found in the appendix.

2 The Unified Policy Framework (UPF)

2.1 The Core of the Unified Policy Framework (UPF)

```
theory  
  UPFCore  
  imports  
    Monads  
begin
```

2.1.1 Foundation

The purpose of this theory is to formalize a somewhat non-standard view on the fundamental concept of a security policy which is worth outlining. This view has arisen from prior experience in the modelling of network (firewall) policies. Instead of regarding policies as relations on resources, sets of permissions, etc., we emphasise the view that a policy is a policy decision function that grants or denies access to resources, permissions, etc. In other words, we model the concrete function that implements the policy decision point in a system, and which represents a "wrapper" around the business logic. An advantage of this view is that it is compatible with many different policy models, enabling a uniform modelling framework to be defined. Furthermore, this function is typically a large cascade of nested conditionals, using conditions referring to an internal state and security contexts of the system or a user. This cascade of conditionals can easily be decomposed into a set of test cases similar to transformations used for binary decision diagrams (BDD), and motivate equivalence class testing for unit test and sequence test scenarios. From the modelling perspective, using HOL as its input language, we will consequently use the expressive power of its underlying functional programming language, including the possibility to define higher-order combinators.

In more detail, we model policies as partial functions based on input data α (arguments, system state, security context, ...) to output data β :

```
datatype 'a decision = allow 'a | deny 'a
```

```
type-synonym ('a,'b) policy = 'a  $\rightarrow$  'b decision (infixr <|-> 0)
```

In the following, we introduce a number of shortcuts and alternative notations. The type of policies is represented as:

```
translations (type) 'a |-> 'b <=> (type) 'a  $\rightarrow$  'b decision  
type-notation policy (infixr <|> 0)
```

... allowing the notation $'\alpha \mapsto '\beta$ for the policy type and the alternative notations for *None* and *Some* of the HOLlibrary $'\alpha$ *option* type:

notation *None* ($\langle \perp \rangle$)

notation *Some* ($\langle [-] \rangle$ 80)

Thus, the range of a policy may consist of $[accept\ x]$ data, of $[deny\ x]$ data, as well as \perp modeling the undefinedness of a policy, i.e. a policy is considered as a partial function. Partial functions are used since we describe elementary policies by partial system behaviour, which are glued together by operators such as function override and functional composition.

We define the two fundamental sets, the allow-set *Allow* and the deny-set *Deny* (written *A* and *D* set for short), to characterize these two main sets of the range of a policy.

definition *Allow* :: ($'\alpha$ *decision*) *set*

where *Allow* = *range allow*

definition *Deny* :: ($'\alpha$ *decision*) *set*

where *Deny* = *range deny*

2.1.2 Policy Constructors

Most elementary policy constructors are based on the update operation *Fun.fun-upd-def* $?f(?a := ?b) = (\lambda x. \text{if } x = ?a \text{ then } ?b \text{ else } ?f\ x)$ and the maplet-notation $a(x \mapsto y)$ used for $a(x \mapsto y)$.

Furthermore, we add notation adopted to our problem domain:

nonterminal *policylets* **and** *policylet*

syntax

-*policylet1* :: [$'a, 'a$] => *policylet* ($\langle - / \mapsto + / - \rangle$)

-*policylet2* :: [$'a, 'a$] => *policylet* ($\langle - / \mapsto - / - \rangle$)

 :: *policylet* => *policylets* ($\langle - \rangle$)

-*Maplets* :: [*policylet*, *policylets*] => *policylets* ($\langle -, / - \rangle$)

-*Maplets* :: [*policylet*, *policylets*] => *policylets* ($\langle -, / - \rangle$)

-*MapUpd* :: [$'a \mid - \rangle 'b$, *policylets*] => $'a \mid - \rangle 'b$ ($\langle - / '(-) \rangle$ [900,0]900)

-*emptypolicy* :: $'a \mid - \rangle 'b$ ($\langle \emptyset \rangle$)

syntax-consts

-*policylet1* \equiv *allow* **and**

-*policylet2* \equiv *deny* **and**

-*Maplets* -*MapUpd* \equiv *fun-upd* **and**

-*emptypolicy* \equiv *Map.empty*

translations

-*MapUpd* *m* (-*Maplets* *xy* *ms*) \equiv -*MapUpd* (-*MapUpd* *m* *xy*) *ms*

-*MapUpd* *m* (-*policylet1* *x* *y*) \equiv $m(x := \text{CONST } \text{Some } (\text{CONST } \text{allow } y))$

$-MapUpd\ m\ (-policylet2\ x\ y) \equiv m(x := CONST\ Some\ (CONST\ deny\ y))$
 $\emptyset \equiv CONST\ Map.empty$

Here are some lemmas essentially showing syntactic equivalences:

lemma *test*: $\emptyset(x \mapsto_+ a, y \mapsto_- b) = \emptyset(x \mapsto_+ a, y \mapsto_- b)$ **by** *simp*

lemma *test2*: $p(x \mapsto_+ a, x \mapsto_- b) = p(x \mapsto_- b)$ **by** *simp*

We inherit a fairly rich theory on policy updates from Map here. Some examples are:

lemma *pol-upd-triv1*: $t\ k = [allow\ x] \implies t(k \mapsto_+ x) = t$
by (*rule ext*) *simp*

lemma *pol-upd-triv2*: $t\ k = [deny\ x] \implies t(k \mapsto_- x) = t$
by (*rule ext*) *simp*

lemma *pol-upd-allow-nonempty*: $t(k \mapsto_+ x) \neq \emptyset$
by *simp*

lemma *pol-upd-deny-nonempty*: $t(k \mapsto_- x) \neq \emptyset$
by *simp*

lemma *pol-upd-eqD1* : $m(a \mapsto_+ x) = n(a \mapsto_+ y) \implies x = y$
by(*auto dest: map-upd-eqD1*)

lemma *pol-upd-eqD2* : $m(a \mapsto_- x) = n(a \mapsto_- y) \implies x = y$
by(*auto dest: map-upd-eqD1*)

lemma *pol-upd-neq1* [*simp*]: $m(a \mapsto_+ x) \neq n(a \mapsto_- y)$
by(*auto dest: map-upd-eqD1*)

2.1.3 Override Operators

Key operators for constructing policies are the override operators. There are four different versions of them, with one of them being the override operator from the Map theory. As it is common to compose policy rules in a “left-to-right-first-fit”-manner, that one is taken as default, defined by a syntax translation from the provided override operator from the Map theory (which does it in reverse order).

syntax

$-policyoverride :: [a \mapsto b, a \mapsto b] \Rightarrow a \mapsto b$ (**infixl** $\langle \oplus \rangle$ 100)

syntax-consts

$-policyoverride \equiv map-add$

translations

$p \oplus q \equiv q ++ p$

Some elementary facts inherited from Map are:

lemma *override-empty*: $p \oplus \emptyset = p$
by *simp*

lemma *empty-override*: $\emptyset \oplus p = p$
by *simp*

lemma *override-assoc*: $p1 \oplus (p2 \oplus p3) = (p1 \oplus p2) \oplus p3$
by *simp*

The following two operators are variants of the standard override. For *override_A*, an allow of wins over a deny. For *override_D*, the situation is dual.

definition *override-A* :: $['\alpha \mapsto '\beta, '\alpha \mapsto '\beta] \Rightarrow '\alpha \mapsto '\beta$ (**infixl** $\langle ++'-A \rangle$ 100)
where $m2 ++-A m1 =$
 $(\lambda x. (case\ m1\ x\ of$
 $\quad [allow\ a] \Rightarrow [allow\ a]$
 $\quad | [deny\ a] \Rightarrow (case\ m2\ x\ of\ [allow\ b] \Rightarrow [allow\ b]$
 $\quad \quad \quad | - \Rightarrow [deny\ a])$
 $\quad | \perp \Rightarrow m2\ x)$
 $)$

syntax

-policyoverride-A :: $['a \mapsto 'b, 'a \mapsto 'b] \Rightarrow 'a \mapsto 'b$ (**infixl** $\langle \oplus_A \rangle$ 100)

syntax-consts

-policyoverride-A \equiv *override-A*

translations

$p \oplus_A q \equiv p ++-A q$

lemma *override-A-empty[simp]*: $p \oplus_A \emptyset = p$
by(*simp add:override-A-def*)

lemma *empty-override-A[simp]*: $\emptyset \oplus_A p = p$
apply (*rule ext*)

apply (*simp add:override-A-def*)

subgoal for x

apply (*case-tac p x*)

apply (*simp-all*)

subgoal for a

apply (*case-tac a*)

apply (*simp-all*)

done

done

done

lemma *override-A-assoc*: $p1 \oplus_A (p2 \oplus_A p3) = (p1 \oplus_A p2) \oplus_A p3$
 by (*rule ext*, *simp add: override-A-def split: decision.splits option.splits*)

definition *override-D* :: $['\alpha \mapsto '\beta, '\alpha \mapsto '\beta] \Rightarrow '\alpha \mapsto '\beta$ (**infixl** $\langle ++'-D \rangle$ 100)

where $m1 ++-D m2 =$

```
( $\lambda x$ . case  $m2$   $x$  of
  | [deny  $a$ ]  $\Rightarrow$  [deny  $a$ ]
  | [allow  $a$ ]  $\Rightarrow$  (case  $m1$   $x$  of [deny  $b$ ]  $\Rightarrow$  [deny  $b$ ]
    | -  $\Rightarrow$  [allow  $a$ ])
  |  $\perp$   $\Rightarrow$   $m1$   $x$ 
)
```

syntax

-policyoverride-D :: $['a \mapsto 'b, 'a \mapsto 'b] \Rightarrow 'a \mapsto 'b$ (**infixl** $\langle \oplus_D \rangle$ 100)

syntax-consts

-policyoverride-D \equiv *override-D*

translations

$p \oplus_D q \equiv p ++-D q$

lemma *override-D-empty[simp]*: $p \oplus_D \emptyset = p$
 by(*simp add: override-D-def*)

lemma *empty-override-D[simp]*: $\emptyset \oplus_D p = p$

apply (*rule ext*)

apply (*simp add: override-D-def*)

subgoal for x

apply (*case-tac* p x , *simp-all*)

subgoal for a

apply (*case-tac* a , *simp-all*)

done

done

done

lemma *override-D-assoc*: $p1 \oplus_D (p2 \oplus_D p3) = (p1 \oplus_D p2) \oplus_D p3$

apply (*rule ext*)

apply (*simp add: override-D-def split: decision.splits option.splits*)

done

2.1.4 Coercion Operators

Often, especially when combining policies of different type, it is necessary to adapt the input or output domain of a policy to a more refined context.

An analogous for the range of a policy is defined as follows:

definition *policy-range-comp* :: [$'\beta \Rightarrow '\gamma, '\alpha \mapsto '\beta$] $\Rightarrow '\alpha \mapsto '\gamma$ (**infixl** $\langle o'-f \rangle$ 55)

where

$$f \text{ o-}f \text{ p} = (\lambda x. \text{ case } p \text{ x of}$$

$$\quad | \text{ [allow } y] \Rightarrow \text{ [allow (f y)]}$$

$$\quad | \text{ [deny } y] \Rightarrow \text{ [deny (f y)]}$$

$$\quad | \perp \Rightarrow \perp)$$

syntax

-policy-range-comp :: [$'\beta \Rightarrow '\gamma, '\alpha \mapsto '\beta$] $\Rightarrow '\alpha \mapsto '\gamma$ (**infixl** $\langle o_f \rangle$ 55)

syntax-consts

-policy-range-comp \equiv *policy-range-comp*

translations

$p \text{ o}_f \text{ q} \equiv p \text{ o-}f \text{ q}$

lemma *policy-range-comp-strict* : $f \text{ o}_f \emptyset = \emptyset$

apply (*rule ext*)

apply (*simp add: policy-range-comp-def*)

done

A generalized version is, where separate coercion functions are applied to the result depending on the decision of the policy is as follows:

definition *range-split* :: [$('\beta \Rightarrow '\gamma) \times ('\beta \Rightarrow '\gamma), '\alpha \mapsto '\beta$] $\Rightarrow '\alpha \mapsto '\gamma$
(**infixr** $\langle \nabla \rangle$ 100)

where $(P) \nabla p = (\lambda x. \text{ case } p \text{ x of}$

$$\quad | \text{ [allow } y] \Rightarrow \text{ [allow ((fst P) y)]}$$

$$\quad | \text{ [deny } y] \Rightarrow \text{ [deny ((snd P) y)]}$$

$$\quad | \perp \quad \quad \Rightarrow \perp)$$

lemma *range-split-strict*[*simp*]: $P \nabla \emptyset = \emptyset$

apply (*rule ext*)

apply (*simp add: range-split-def*)

done

lemma *range-split-charn*:

$$(f, g) \nabla p = (\lambda x. \text{ case } p \text{ x of}$$

$$\quad | \text{ [allow } x] \Rightarrow \text{ [allow (f x)]}$$

$$\quad | \text{ [deny } x] \Rightarrow \text{ [deny (g x)]}$$

$$\quad | \perp \quad \quad \Rightarrow \perp)$$

apply (*simp add: range-split-def*)

apply (*rule ext*)

subgoal for x

apply (*case-tac p x*)

apply (*simp-all*)

```

subgoal for a
  apply (case-tac a)
  apply (simp-all)
  done
done
done

```

The connection between these two becomes apparent if considering the following lemma:

```

lemma range-split-vs-range-compose: (f,f)  $\nabla$  p = f of p
  by(simp add: range-split-charn policy-range-comp-def)

```

```

lemma range-split-id [simp]: (id,id)  $\nabla$  p = p
  apply (rule ext)
  apply (simp add: range-split-charn id-def)
  subgoal for x
    apply (case-tac p x)
    apply (simp-all)
    subgoal for a
      apply (case-tac a)
      apply (simp-all)
      done
    done
  done

```

```

lemma range-split-bi-compose [simp]: (f1,f2)  $\nabla$  (g1,g2)  $\nabla$  p = (f1 o g1,f2 o g2)  $\nabla$  p
  apply (rule ext)
  apply (simp add: range-split-charn comp-def)
  subgoal for x
    apply (case-tac p x)
    apply (simp-all)
    subgoal for a
      apply (case-tac a)
      apply (simp-all)
      done
    done
  done

```

The next three operators are rather exotic and in most cases not used.

The following is a variant of `range_split`, where the change in the decision depends on the input instead of the output.

```

definition dom-split2a :: [('α  $\rightarrow$  'γ)  $\times$  ('α  $\rightarrow$  'γ), 'α  $\mapsto$  'β]  $\Rightarrow$  'α  $\mapsto$  'γ      (infixr
<Δa> 100)
where P Δa p = ( $\lambda x$ . case p x of

```

$$\begin{aligned}
& \lfloor \text{allow } y \rfloor \Rightarrow \lfloor \text{allow } (\text{the } ((\text{fst } P) x)) \rfloor \\
& \lfloor \text{deny } y \rfloor \Rightarrow \lfloor \text{deny } (\text{the } ((\text{snd } P) x)) \rfloor \\
& \lfloor \perp \rfloor \Rightarrow \perp
\end{aligned}$$

definition *dom-split2* :: $[('\alpha \Rightarrow '\gamma) \times ('\alpha \Rightarrow '\gamma), '\alpha \mapsto '\beta] \Rightarrow '\alpha \mapsto '\gamma$ (**infixr** $\langle \Delta \rangle$ 100)

where $P \Delta p = (\lambda x. \text{ case } p \ x \ \text{of}$
 $\quad \lfloor \text{allow } y \rfloor \Rightarrow \lfloor \text{allow } ((\text{fst } P) x) \rfloor$
 $\quad \lfloor \text{deny } y \rfloor \Rightarrow \lfloor \text{deny } ((\text{snd } P) x) \rfloor$
 $\quad \lfloor \perp \rfloor \Rightarrow \perp)$

definition *range-split2* :: $[('\alpha \Rightarrow '\gamma) \times ('\alpha \Rightarrow '\gamma), '\alpha \mapsto '\beta] \Rightarrow '\alpha \mapsto (''\beta \times '\gamma)$ (**infixr** $\langle \nabla 2 \rangle$ 100)

where $P \nabla 2 p = (\lambda x. \text{ case } p \ x \ \text{of}$
 $\quad \lfloor \text{allow } y \rfloor \Rightarrow \lfloor \text{allow } (y, (\text{fst } P) x) \rfloor$
 $\quad \lfloor \text{deny } y \rfloor \Rightarrow \lfloor \text{deny } (y, (\text{snd } P) x) \rfloor$
 $\quad \lfloor \perp \rfloor \Rightarrow \perp)$

The following operator is used for transition policies only: a transition policy is transformed into a state-exception monad. Such a monad can for example be used for test case generation using HOL-Testgen [9].

definition *policy2MON* :: $('i \times '\sigma \mapsto 'o \times '\sigma) \Rightarrow ('i \Rightarrow ('o \ \text{decision}, '\sigma)) \text{ MON}_{SE}$

where $\text{policy2MON } p = (\lambda \iota \ \sigma. \text{ case } p \ (\iota, \sigma) \ \text{of}$
 $\quad \lfloor (\text{allow } (\text{outs}, \sigma')) \rfloor \Rightarrow \lfloor (\text{allow } \text{outs}, \sigma') \rfloor$
 $\quad \lfloor (\text{deny } (\text{outs}, \sigma')) \rfloor \Rightarrow \lfloor (\text{deny } \text{outs}, \sigma') \rfloor$
 $\quad \lfloor \perp \rfloor \Rightarrow \perp)$

lemmas *UPFCoreDefs* = *Allow-def Deny-def override-A-def override-D-def policy-range-comp-def*

range-split-def dom-split2-def map-add-def restrict-map-def

end

2.2 Elementary Policies

theory

ElementaryPolicies

imports

UPFCore

begin

In this theory, we introduce the elementary policies of UPF that build the basis for more complex policies. These complex policies, respectively, embedding of well-known access control or security models, are build by composing the elementary policies defined in this theory.

2.2.1 The Core Policy Combinators: Allow and Deny Everything

definition

$deny\text{-}pfun \quad :: ('\alpha \multimap '\beta) \Rightarrow ('\alpha \mapsto '\beta) (\langle AllD \rangle)$

where

$$deny\text{-}pfun\ pf \equiv (\lambda x. \text{case } pf\ x\ \text{of} \\ \quad [y] \Rightarrow [deny\ (y)] \\ \quad |\perp \Rightarrow \perp)$$

definition

$allow\text{-}pfun \quad :: ('\alpha \multimap '\beta) \Rightarrow ('\alpha \mapsto '\beta) (\langle AllA \rangle)$

where

$$allow\text{-}pfun\ pf \equiv (\lambda x. \text{case } pf\ x\ \text{of} \\ \quad [y] \Rightarrow [allow\ (y)] \\ \quad |\perp \Rightarrow \perp)$$

syntax

$\text{-}allow\text{-}pfun \quad :: ('\alpha \multimap '\beta) \Rightarrow ('\alpha \mapsto '\beta) (\langle A_p \rangle)$

syntax-consts

$\text{-}allow\text{-}pfun \equiv allow\text{-}pfun$

translations

$A_p\ f \equiv AllA\ f$

syntax

$\text{-}deny\text{-}pfun \quad :: ('\alpha \multimap '\beta) \Rightarrow ('\alpha \mapsto '\beta) (\langle D_p \rangle)$

syntax-consts

$\text{-}deny\text{-}pfun \equiv deny\text{-}pfun$

translations

$D_p\ f \equiv AllD\ f$

notation

$deny\text{-}pfun$ (**binder** $\langle \forall D \rangle$ 10) **and**

$allow\text{-}pfun$ (**binder** $\langle \forall A \rangle$ 10)

lemma $AllD\text{-}norm[simp]$: $deny\text{-}pfun\ (id\ o\ (\lambda x. [x])) = (\forall Dx. [x])$

by($simp\ add:id\text{-}def\ comp\text{-}def$)

lemma $AllD\text{-}norm2[simp]$: $deny\text{-}pfun\ (Some\ o\ id) = (\forall Dx. [x])$

by($simp\ add:id\text{-}def\ comp\text{-}def$)

lemma $AllA\text{-}norm[simp]$: $allow\text{-}pfun\ (id\ o\ Some) = (\forall Ax. [x])$

by($simp\ add:id\text{-}def\ comp\text{-}def$)

lemma $AllA\text{-}norm2[simp]$: $allow\text{-}pfun\ (Some\ o\ id) = (\forall Ax. [x])$

by(*simp add:id-def comp-def*)

lemma *AllA-apply[simp]*: $(\forall Ax. \text{Some } (P x)) x = \lfloor \text{allow } (P x) \rfloor$
by(*simp add:allow-pfun-def*)

lemma *AllD-apply[simp]*: $(\forall Dx. \text{Some } (P x)) x = \lfloor \text{deny } (P x) \rfloor$
by(*simp add:deny-pfun-def*)

lemma *neq-Allow-Deny*: $pf \neq \emptyset \implies (\text{deny-pfun } pf) \neq (\text{allow-pfun } pf)$
apply (*erule contrapos-nn*)
apply (*rule ext*)
subgoal for *x*
 apply (*drule-tac x=x in fun-cong*)
 apply (*auto simp: deny-pfun-def allow-pfun-def*)
 apply (*case-tac pf x = \perp*)
 apply (*auto*)
 done
done

2.2.2 Common Instances

definition *allow-all-fun* :: $('α \Rightarrow 'β) \Rightarrow ('α \mapsto 'β) (\langle A_f \rangle)$
 where *allow-all-fun* *f* = *allow-pfun* (*Some o f*)

definition *deny-all-fun* :: $('α \Rightarrow 'β) \Rightarrow ('α \mapsto 'β) (\langle D_f \rangle)$
 where *deny-all-fun* *f* \equiv *deny-pfun* (*Some o f*)

definition
 deny-all-id :: $'α \mapsto 'α (\langle D_I \rangle)$ **where**
 deny-all-id \equiv *deny-pfun* (*id o Some*)

definition
 allow-all-id :: $'α \mapsto 'α (\langle A_I \rangle)$ **where**
 allow-all-id \equiv *allow-pfun* (*id o Some*)

definition
 allow-all :: $('α \mapsto \text{unit}) (\langle A_U \rangle)$ **where**
 allow-all *p* = $\lfloor \text{allow } () \rfloor$

definition
 deny-all :: $('α \mapsto \text{unit}) (\langle D_U \rangle)$ **where**
 deny-all *p* = $\lfloor \text{deny } () \rfloor$

... and resulting properties:

lemma $A_I \oplus \text{Map.empty} = A_I$
 by *simp*

lemma $A_f f \oplus \text{Map.empty} = A_f f$
 by *simp*

lemma $\text{allow-pfun Map.empty} = \text{Map.empty}$
 apply (*rule ext*)
 apply (*simp add: allow-pfun-def*)
 done

lemma $\text{allow-left-cancel} : \text{dom pf} = \text{UNIV} \implies (\text{allow-pfun pf}) \oplus x = (\text{allow-pfun pf})$

apply (*rule ext*)
 apply (*auto simp: allow-pfun-def option.splits*)
 done

lemma $\text{deny-left-cancel} : \text{dom pf} = \text{UNIV} \implies (\text{deny-pfun pf}) \oplus x = (\text{deny-pfun pf})$

apply (*rule ext*)
 by (*auto simp: deny-pfun-def option.splits*)

2.2.3 Domain, Range, and Restrictions

Since policies are essentially maps, we inherit the basic definitions for domain and range on Maps:

$\text{Map.dom_def} : \text{dom } ?m = \{a. ?m a \neq \perp\}$

whereas range is just an abbreviation for image:

```
abbreviation range :: "('a => 'b) => 'b set"
  where -- "of function" "range f == f ` UNIV"
```

As a consequence, we inherit the following properties on policies:

- $\text{Map.domD } ?a \in \text{dom } ?m \implies \exists b. ?m ?a = [b]$
- $\text{Map.domI } ?m ?a = [?b] \implies ?a \in \text{dom } ?m$
- $\text{Map.domIff } (?a \in \text{dom } ?m) = (?m ?a \neq \perp)$
- $\text{Map.dom_const dom } (\lambda x. [?f x]) = \text{UNIV}$
- $\text{Map.dom_def dom } ?m = \{a. ?m a \neq \perp\}$
- $\text{Map.dom_empty dom } (\lambda x. \perp) = \{\}$
- $\text{Map.dom_eq_empty_conv } (\text{dom } ?f = \{\}) = (?f = (\lambda x. \perp))$

- $\text{Map.dom_eq_singleton_conv } (dom \ ?f = \{?x\}) = (\exists v. \ ?f = [?x \mapsto v])$
- $\text{Map.dom_fun_upd } dom \ (?f(?x := ?y)) = (if \ ?y = \perp \text{ then } dom \ ?f - \{?x\} \text{ else } insert \ ?x \ (dom \ ?f))$
- $\text{Map.dom_if } dom \ (\lambda x. \ if \ ?P \ x \ \text{then} \ ?f \ x \ \text{else} \ ?g \ x) = dom \ ?f \cap \{x. \ ?P \ x\} \cup dom \ ?g \cap \{x. \ \neg \ ?P \ x\}$
- $\text{Map.dom_map_add } dom \ (?n \oplus \ ?m) = dom \ ?n \cup dom \ ?m$

However, some properties are specific to policy concepts:

lemma *sub-ran* : $ran \ p \subseteq Allow \cup Deny$

```

apply (auto simp: Allow-def Deny-def ran-def full-SetCompr-eq[symmetric])[1]
subgoal for  $x \ a$ 
  apply (case-tac  $x$ )
  apply (simp-all)
done
done

```

lemma *dom-allow-pfun* [simp]: $dom(allow\text{-}pfun \ f) = dom \ f$

```

apply (auto simp: allow-pfun-def)
subgoal for  $x \ y$ 
  apply (case-tac  $f \ x, \ simp\text{-}all$ )
done
done

```

lemma *dom-allow-all*: $dom(A_f \ f) = UNIV$

```

by(auto simp: allow-all-fun-def o-def)

```

lemma *dom-deny-pfun* [simp]: $dom(deny\text{-}pfun \ f) = dom \ f$

```

apply (auto simp: deny-pfun-def)[1]
apply (case-tac  $f \ x$ )
apply (simp-all)
done

```

lemma *dom-deny-all*: $dom(D_f \ f) = UNIV$

```

by(auto simp: deny-all-fun-def o-def)

```

lemma *ran-allow-pfun* [simp]: $ran(allow\text{-}pfun \ f) = allow \ (ran \ f)$

```

apply (simp add: allow-pfun-def ran-def)
apply (rule set-eqI)
apply (auto)[1]
subgoal for  $x \ a$ 
  apply (case-tac  $f \ a$ )

```

```

  apply (auto simp: image-def)[1]
  apply (auto simp: image-def)[1]
done
subgoal for xa a
  apply (rule-tac x=a in exI)
  apply (simp)
  done
done

```

```

lemma ran-allow-all: ran(Af id) = Allow
  apply (simp add: allow-all-fun-def Allow-def o-def)
  apply (rule set-eqI)
  apply (auto simp: image-def ran-def)
  done

```

```

lemma ran-deny-pfun[simp]: ran(deny-pfun f) = deny ‘ (ran f)
  apply (simp add: deny-pfun-def ran-def)
  apply (rule set-eqI)
  apply (auto)[1]
  subgoal for x a
    apply (case-tac f a)
    apply (auto simp: image-def)[1]
    apply (auto simp: image-def)[1]
    done
  subgoal for xa a
    apply (rule-tac x=a in exI)
    apply (simp)
    done
  done

```

```

lemma ran-deny-all: ran(Df id) = Deny
  apply (simp add: deny-all-fun-def Deny-def o-def)
  apply (rule set-eqI)
  apply (auto simp: image-def ran-def)
  done

```

Reasoning over `dom` is most crucial since it paves the way for simplification and reordering of policies composed by `override` (i.e. by the normal left-to-right rule composition method).

- $\text{Map.dom_map_add dom } (?n \oplus ?m) = \text{dom } ?n \cup \text{dom } ?m$
- $\text{Map.inj_on_map_add_dom inj-on } (?m' \oplus ?m) (\text{dom } ?m') = \text{inj-on } ?m' (\text{dom } ?m')$
- $\text{Map.map_add_comm dom } ?m1.0 \cap \text{dom } ?m2.0 = \{\} \implies ?m2.0 \oplus ?m1.0 =$

$$?m1.0 \oplus ?m2.0$$

- `Map.map_add_dom_app_simps(1)` $?m \in \text{dom } ?l2.0 \implies (?l2.0 \oplus ?l1.0) ?m = ?l2.0 ?m$
- `Map.map_add_dom_app_simps(2)` $?m \notin \text{dom } ?l1.0 \implies (?l2.0 \oplus ?l1.0) ?m = ?l2.0 ?m$
- `Map.map_add_dom_app_simps(3)` $?m \notin \text{dom } ?l2.0 \implies (?l2.0 \oplus ?l1.0) ?m = ?l1.0 ?m$
- `Map.map_add_upd_left` $?m \notin \text{dom } ?e2.0 \implies ?e2.0 \oplus ?e1.0(?m \mapsto ?u1.0) = (?e2.0 \oplus ?e1.0)(?m \mapsto ?u1.0)$

The latter rule also applies to allow- and deny-override.

definition `dom-restrict` :: $['\alpha \text{ set}, '\alpha \mapsto '\beta] \Rightarrow '\alpha \mapsto '\beta$ (**infixr** \triangleleft 55)

where $S \triangleleft p \equiv (\lambda x. \text{if } x \in S \text{ then } p \text{ else } \perp)$

lemma `dom-dom-restrict[simp]` : $\text{dom}(S \triangleleft p) = S \cap \text{dom } p$

apply (`auto simp: dom-restrict-def`)

subgoal for $x \ y$

apply (`case-tac x \in S`)

apply (`simp-all`)

done

subgoal for $x \ y$

apply (`case-tac x \in S`)

apply (`simp-all`)

done

done

lemma `dom-restrict-idem[simp]` : $(\text{dom } p) \triangleleft p = p$

apply (`rule ext`)

apply (`auto simp: dom-restrict-def`)

dest: neq-commute[THEN iffD1, THEN not-None-eq [THEN iffD1]]

done

lemma `dom-restrict-inter[simp]` : $T \triangleleft S \triangleleft p = T \cap S \triangleleft p$

apply (`rule ext`)

apply (`auto simp: dom-restrict-def`)

dest: neq-commute[THEN iffD1, THEN not-None-eq [THEN iffD1]]

done

definition `ran-restrict` :: $['\alpha \mapsto '\beta, '\beta \text{ decision set}] \Rightarrow '\alpha \mapsto '\beta$ (**infixr** \triangleright 55)

where $p \triangleright S \equiv (\lambda x. \text{if } p \ x \in (\text{Some } S) \text{ then } p \text{ else } \perp)$

definition *ran-restrict2* :: [$'\alpha \mapsto '\beta, '\beta$ decision set] $\Rightarrow '\alpha \mapsto '\beta$ (**infixr** $\triangleright 2$ 55)
where $p \triangleright 2 S \equiv (\lambda x. \text{if } (\text{the } (p \ x)) \in (S) \text{ then } p \ x \text{ else } \perp)$

lemma *ran-restrict = ran-restrict2*
apply (*rule ext*)
apply (*simp add: ran-restrict-def ran-restrict2-def*)
subgoal for $x \ xa \ xb$
apply (*case-tac x xb*)
apply *simp-all*
apply (*metis inj-Some inj-image-mem-iff*)
done
done

lemma *ran-ran-restrict[simp]* : $\text{ran}(p \triangleright S) = S \cap \text{ran } p$
by(*auto simp: ran-restrict-def image-def ran-def*)

lemma *ran-restrict-idem[simp]* : $p \triangleright (\text{ran } p) = p$
apply (*rule ext*)
apply (*auto simp: ran-restrict-def image-def Ball-def ran-def*)
apply (*erule contrapos-pp*)
apply (*auto dest!: neq-commute[THEN iffD1, THEN not-None-eq [THEN iffD1]]*)
done

lemma *ran-restrict-inter[simp]* : $(p \triangleright S) \triangleright T = p \triangleright T \cap S$
apply (*rule ext*)
apply (*auto simp: ran-restrict-def*
dest: neq-commute[THEN iffD1, THEN not-None-eq [THEN iffD1]])
done

lemma *ran-gen-A[simp]* : $(\forall Ax. \lceil P \ x \rceil) \triangleright \text{Allow} = (\forall Ax. \lceil P \ x \rceil)$
apply (*rule ext*)
apply (*auto simp: Allow-def ran-restrict-def*)
done

lemma *ran-gen-D[simp]* : $(\forall Dx. \lceil P \ x \rceil) \triangleright \text{Deny} = (\forall Dx. \lceil P \ x \rceil)$
apply (*rule ext*)
apply (*auto simp: Deny-def ran-restrict-def*)
done

lemmas *ElementaryPoliciesDefs* = *deny-pfun-def allow-pfun-def allow-all-fun-def*
deny-all-fun-def

allow-all-id-def deny-all-id-def allow-all-def deny-all-def
dom-restrict-def ran-restrict-def

end

2.3 Sequential Composition

theory

SeqComposition

imports

ElementaryPolicies

begin

Sequential composition is based on the idea that two policies are to be combined by applying the second policy to the output of the first one. Again, there are four possibilities how the decisions can be combined.

2.3.1 Flattening

A key concept of sequential policy composition is the flattening of nested decisions. There are four possibilities, and these possibilities will give the various flavours of policy composition.

fun *flat-orA* :: (' α decision) decision \Rightarrow (' α decision)

where *flat-orA*(*allow*(*allow* *y*)) = *allow* *y*

|*flat-orA*(*allow*(*deny* *y*)) = *allow* *y*

|*flat-orA*(*deny*(*allow* *y*)) = *allow* *y*

|*flat-orA*(*deny*(*deny* *y*)) = *deny* *y*

lemma *flat-orA-deny*[*dest*]: *flat-orA* *x* = *deny* *y* \Longrightarrow *x* = *deny*(*deny* *y*)

apply (*case-tac* *x*)

apply (*rename-tac* α)

apply (*case-tac* α , *simp-all*)[1]

apply (*rename-tac* α)

apply (*case-tac* α , *simp-all*)[1]

done

lemma *flat-orA-allow*[*dest*]: *flat-orA* *x* = *allow* *y* \Longrightarrow *x* = *allow*(*allow* *y*)

\vee *x* = *allow*(*deny* *y*)

\vee *x* = *deny*(*allow* *y*)

apply (*case-tac* *x*)

apply (*rename-tac* α)

apply (*case-tac* α , *simp-all*)[1]

apply (*rename-tac* α)

apply (*case-tac* α , *simp-all*)[1]

done

```

fun flat-orD :: ('α decision) decision ⇒ ('α decision)
where flat-orD(allow(allow y)) = allow y
      |flat-orD(allow(deny y)) = deny y
      |flat-orD(deny(allow y)) = deny y
      |flat-orD(deny(deny y)) = deny y

```

```

lemma flat-orD-allow[dest]: flat-orD x = allow y ⇒ x = allow(allow y)
apply (case-tac x)
apply (rename-tac α)
apply (case-tac α, simp-all)[1]
apply (rename-tac α)
apply (case-tac α, simp-all)[1]
done

```

```

lemma flat-orD-deny[dest]: flat-orD x = deny y ⇒ x = deny(deny y)
      ∨ x = allow(deny y)
      ∨ x = deny(allow y)

apply (case-tac x)
apply (rename-tac α)
apply (case-tac α, simp-all)[1]
apply (rename-tac α)
apply (case-tac α, simp-all)[1]
done

```

```

fun flat-1 :: ('α decision) decision ⇒ ('α decision)
where flat-1(allow(allow y)) = allow y
      |flat-1(allow(deny y)) = allow y
      |flat-1(deny(allow y)) = deny y
      |flat-1(deny(deny y)) = deny y

```

```

lemma flat-1-allow[dest]: flat-1 x = allow y ⇒ x = allow(allow y) ∨ x = allow(deny
y)
apply (case-tac x)
apply (rename-tac α)
apply (case-tac α, simp-all)[1]
apply (rename-tac α)
apply (case-tac α, simp-all)[1]
done

```

```

lemma flat-1-deny[dest]: flat-1 x = deny y ⇒ x = deny(deny y) ∨ x = deny(allow
y)
apply (case-tac x)
apply (rename-tac α)

```

```

apply (case-tac  $\alpha$ , simp-all)[1]
apply (rename-tac  $\alpha$ )
apply (case-tac  $\alpha$ , simp-all)[1]
done

```

```

fun flat-2 :: (' $\alpha$  decision) decision  $\Rightarrow$  (' $\alpha$  decision)
where flat-2(allow(allow y)) = allow y
      |flat-2(allow(deny y)) = deny y
      |flat-2(deny(allow y)) = allow y
      |flat-2(deny(deny y)) = deny y

```

```

lemma flat-2-allow[dest]: flat-2 x = allow y  $\Longrightarrow$  x = allow(allow y)  $\vee$  x = deny(allow y)
apply (case-tac x)
apply (rename-tac  $\alpha$ )
apply (case-tac  $\alpha$ , simp-all)[1]
apply (rename-tac  $\alpha$ )
apply (case-tac  $\alpha$ , simp-all)[1]
done

```

```

lemma flat-2-deny[dest]: flat-2 x = deny y  $\Longrightarrow$  x = deny(deny y)  $\vee$  x = allow(deny y)
apply (case-tac x)
apply (rename-tac  $\alpha$ )
apply (case-tac  $\alpha$ , simp-all)[1]
apply (rename-tac  $\alpha$ )
apply (case-tac  $\alpha$ , simp-all)[1]
done

```

2.3.2 Policy Composition

The following definition allows to compose two policies. Denies and allows are transferred.

```

fun lift :: (' $\alpha$   $\mapsto$  ' $\beta$ )  $\Rightarrow$  (' $\alpha$  decision  $\mapsto$  ' $\beta$  decision)
where lift f (deny s) = (case f s of
      |y]  $\Rightarrow$  |deny y]
      |  $\perp$   $\Rightarrow$   $\perp$ )
  | lift f (allow s) = (case f s of
      |y]  $\Rightarrow$  |allow y]
      |  $\perp$   $\Rightarrow$   $\perp$ )

```

```

lemma lift-mt [simp]: lift  $\emptyset$  =  $\emptyset$ 
apply (rule ext)
subgoal for x

```

```

apply (case-tac x)
  apply (simp-all)
done
done

```

Since policies are maps, we inherit a composition on them. However, this results in nestings of decisions—which must be flattened. As we now that there are four different forms of flattening, we have four different forms of policy composition:

definition

```

comp-orA :: ['β→'γ, 'α→'β] ⇒ 'α→'γ (infixl <o'-orA> 55) where
p2 o-orA p1 ≡ (map-option flat-orA) o (lift p2 om p1)

```

notation

```

comp-orA (infixl <ovA> 55)

```

```

lemma comp-orA-mt[simp]:p ovA  $\emptyset = \emptyset$ 
by(simp add: comp-orA-def)

```

```

lemma mt-comp-orA[simp]: $\emptyset$  ovA p =  $\emptyset$ 
by(simp add: comp-orA-def)

```

definition

```

comp-orD :: ['β→'γ, 'α→'β] ⇒ 'α→'γ (infixl <o'-orD> 55) where
p2 o-orD p1 ≡ (map-option flat-orD) o (lift p2 om p1)

```

notation

```

comp-orD (infixl <oorD> 55)

```

```

lemma comp-orD-mt[simp]:p o-orD  $\emptyset = \emptyset$ 
by(simp add: comp-orD-def)

```

```

lemma mt-comp-orD[simp]: $\emptyset$  o-orD p =  $\emptyset$ 
by(simp add: comp-orD-def)

```

definition

```

comp-1 :: ['β→'γ, 'α→'β] ⇒ 'α→'γ (infixl <o'-1> 55) where
p2 o-1 p1 ≡ (map-option flat-1) o (lift p2 om p1)

```

notation

```

comp-1 (infixl <o1> 55)

```

```

lemma comp-1-mt[simp]:p o1  $\emptyset = \emptyset$ 
by(simp add: comp-1-def)

```

lemma *mt-comp-1*[*simp*]: $\emptyset \circ_1 p = \emptyset$
by(*simp add: comp-1-def*)

definition

comp-2 :: [$'\beta \mapsto '\gamma, '\alpha \mapsto '\beta$] $\Rightarrow '\alpha \mapsto '\gamma$ (**infixl** $\langle o'-2 \rangle$ 55) **where**
p2 o-2 p1 \equiv (*map-option flat-2*) *o* (*lift p2* \circ_m *p1*)

notation

comp-2 (**infixl** $\langle \circ_2 \rangle$ 55)

lemma *comp-2-mt*[*simp*]: $p \circ_2 \emptyset = \emptyset$
by(*simp add: comp-2-def*)

lemma *mt-comp-2*[*simp*]: $\emptyset \circ_2 p = \emptyset$
by(*simp add: comp-2-def*)

end

2.4 Parallel Composition

theory

ParallelComposition

imports

ElementaryPolicies

begin

The following combinators are based on the idea that two policies are executed in parallel. Since both input and the output can differ, we chose to pair them.

The new input pair will often contain repetitions, which can be reduced using the domain-restriction and domain-reduction operators. Using additional range-modifying operators such as ∇ , decide which result argument is chosen; this might be the first or the latter or, in case that $\beta = \gamma$, and β underlies a lattice structure, the supremum or infimum of both, or, an arbitrary combination of them.

In any case, although we have strictly speaking a pairing of decisions and not a nesting of them, we will apply the same notational conventions as for the latter, i.e. as for flattening.

2.4.1 Parallel Combinators: Foundations

There are four possible semantics how the decision can be combined, thus there are four parallel composition operators. For each of them, we prove several properties.

definition *prod-orA* :: [$'\alpha \mapsto '\beta, '\gamma \mapsto '\delta$] $\Rightarrow (' \alpha \times '\gamma \mapsto '\beta \times '\delta)$ (**infixr** $\langle \otimes_{\vee A} \rangle$ 55)
where $p1 \otimes_{\vee A} p2 =$

```

( $\lambda(x,y).$  (case  $p1$   $x$  of
  [allow  $d1$ ]  $\Rightarrow$ (case  $p2$   $y$  of
    [allow  $d2$ ]  $\Rightarrow$  [allow( $d1,d2$ )]
    | [deny  $d2$ ]  $\Rightarrow$  [allow( $d1,d2$ )]
    |  $\perp \Rightarrow \perp$ )
  | [deny  $d1$ ]  $\Rightarrow$ (case  $p2$   $y$  of
    [allow  $d2$ ]  $\Rightarrow$  [allow( $d1,d2$ )]
    | [deny  $d2$ ]  $\Rightarrow$  [deny ( $d1,d2$ )]
    |  $\perp \Rightarrow \perp$ )
  |  $\perp \Rightarrow \perp$ ))

```

```

lemma prod-orA-mt[simp]: $p \otimes_{\vee A} \emptyset = \emptyset$ 
  apply (rule ext)
  apply (simp add: prod-orA-def)
  apply (auto)
  apply (simp split: option.splits decision.splits)
  done

```

```

lemma mt-prod-orA[simp]: $\emptyset \otimes_{\vee A} p = \emptyset$ 
  apply (rule ext)
  apply (simp add: prod-orA-def)
  done

```

```

lemma prod-orA-quasi-commute:  $p2 \otimes_{\vee A} p1 = (((\lambda(x,y). (y,x)) \text{ o-f } (p1 \otimes_{\vee A} p2)))$ 
o ( $\lambda(a,b).(b,a)$ )
  apply (rule ext)
  apply (simp add: prod-orA-def policy-range-comp-def o-def)
  apply (auto)[1]
  apply (simp split: option.splits decision.splits)
  done

```

```

definition prod-orD :: [ $'\alpha \mapsto '\beta, '\gamma \mapsto '\delta$ ]  $\Rightarrow$  ( $'\alpha \times '\gamma \mapsto '\beta \times '\delta$ ) (infixr  $\langle \otimes_{\vee D} \rangle$ 
55)

```

```

where  $p1 \otimes_{\vee D} p2 =$ 
  ( $\lambda(x,y).$  (case  $p1$   $x$  of
    [allow  $d1$ ]  $\Rightarrow$ (case  $p2$   $y$  of
      [allow  $d2$ ]  $\Rightarrow$  [allow( $d1,d2$ )]
      | [deny  $d2$ ]  $\Rightarrow$  [deny( $d1,d2$ )]
      |  $\perp \Rightarrow \perp$ )
    | [deny  $d1$ ]  $\Rightarrow$ (case  $p2$   $y$  of
      [allow  $d2$ ]  $\Rightarrow$  [deny( $d1,d2$ )]
      | [deny  $d2$ ]  $\Rightarrow$  [deny ( $d1,d2$ )]
      |  $\perp \Rightarrow \perp$ )
    |  $\perp \Rightarrow \perp$ ))

```

```

lemma prod-orD-mt[simp]:  $p \otimes_{\vee D} \emptyset = \emptyset$ 
  apply (rule ext)
  apply (simp add: prod-orD-def)
  apply (auto)[1]
  apply (simp split: option.splits decision.splits)
  done

```

```

lemma mt-prod-orD[simp]:  $\emptyset \otimes_{\vee D} p = \emptyset$ 
  apply (rule ext)
  apply (simp add: prod-orD-def)
  done

```

```

lemma prod-orD-quasi-commute:  $p2 \otimes_{\vee D} p1 = (((\lambda(x,y). (y,x)) \circ f (p1 \otimes_{\vee D} p2)))$ 
   $\circ (\lambda(a,b).(b,a))$ 
  apply (rule ext)
  apply (simp add: prod-orD-def policy-range-comp-def o-def)
  apply (auto)[1]
  apply (simp split: option.splits decision.splits)
  done

```

The following two combinators are by definition non-commutative, but still strict.

```

definition prod-1 :: [ $'\alpha \mapsto '\beta$ ,  $'\gamma \mapsto '\delta$ ]  $\Rightarrow$  ( $'\alpha \times '\gamma \mapsto '\beta \times '\delta$ ) (infixr  $\langle \otimes_1 \rangle$  55)
  where  $p1 \otimes_1 p2 \equiv$ 
    ( $\lambda(x,y). (case\ p1\ x\ of$ 
      [allow d1]  $\Rightarrow$  (case p2 y of
        [allow d2]  $\Rightarrow$  [allow(d1,d2)]
        | [deny d2]  $\Rightarrow$  [allow(d1,d2)]
        |  $\perp \Rightarrow \perp$ )
      | [deny d1]  $\Rightarrow$  (case p2 y of
        [allow d2]  $\Rightarrow$  [deny(d1,d2)]
        | [deny d2]  $\Rightarrow$  [deny(d1,d2)]
        |  $\perp \Rightarrow \perp$ )
      |  $\perp \Rightarrow \perp$ ))

```

```

lemma prod-1-mt[simp]:  $p \otimes_1 \emptyset = \emptyset$ 
  apply (rule ext)
  apply (simp add: prod-1-def)
  apply (auto)[1]
  apply (simp split: option.splits decision.splits)
  done

```

```

lemma mt-prod-1[simp]:  $\emptyset \otimes_1 p = \emptyset$ 
  apply (rule ext)

```

apply (*simp add: prod-1-def*)
done

definition *prod-2* :: [$'\alpha \mapsto '\beta, '\gamma \mapsto '\delta$] \Rightarrow ($'\alpha \times '\gamma \mapsto '\beta \times '\delta$) (**infixr** $\langle \otimes_2 \rangle$ 55)

where $p1 \otimes_2 p2 \equiv$

($\lambda(x,y).$ (*case p1 x of*
 [*allow d1*] \Rightarrow (*case p2 y of*
 [*allow d2*] \Rightarrow [*allow*(*d1*,*d2*)]
 | [*deny d2*] \Rightarrow [*deny* (*d1*,*d2*)]
 | $\perp \Rightarrow \perp$)
 | [*deny d1*] \Rightarrow (*case p2 y of*
 [*allow d2*] \Rightarrow [*allow*(*d1*,*d2*)]
 | [*deny d2*] \Rightarrow [*deny* (*d1*,*d2*)]
 | $\perp \Rightarrow \perp$)
 | $\perp \Rightarrow \perp$))

lemma *prod-2-mt[simp]:p* $\otimes_2 \emptyset = \emptyset$

apply (*rule ext*)
apply (*simp add: prod-2-def*)
apply (*auto*)[1]
apply (*simp split: option.splits decision.splits*)
done

lemma *mt-prod-2[simp]:* $\emptyset \otimes_2 p = \emptyset$

apply (*rule ext*)
apply (*simp add: prod-2-def*)
done

definition *prod-1-id* :: [$'\alpha \mapsto '\beta, '\alpha \mapsto '\gamma$] \Rightarrow ($'\alpha \mapsto '\beta \times '\gamma$) (**infixr** $\langle \otimes_{1I} \rangle$ 55)

where $p \otimes_{1I} q = (p \otimes_1 q) \circ (\lambda x. (x,x))$

lemma *prod-1-id-mt[simp]:p* $\otimes_{1I} \emptyset = \emptyset$

apply (*rule ext*)
apply (*simp add: prod-1-id-def*)
done

lemma *mt-prod-1-id[simp]:* $\emptyset \otimes_{1I} p = \emptyset$

apply (*rule ext*)
apply (*simp add: prod-1-id-def prod-1-def*)
done

definition *prod-2-id* :: [$'\alpha \mapsto '\beta, '\alpha \mapsto '\gamma$] \Rightarrow ($'\alpha \mapsto '\beta \times '\gamma$) (**infixr** $\langle \otimes_{2I} \rangle$ 55)

where $p \otimes_{2I} q = (p \otimes_2 q) \circ (\lambda x. (x,x))$

```

lemma prod-2-id-mt[simp]:  $p \otimes_{2I} \emptyset = \emptyset$ 
  apply (rule ext)
  apply (simp add: prod-2-id-def)
  done

```

```

lemma mt-prod-2-id[simp]:  $\emptyset \otimes_{2I} p = \emptyset$ 
  apply (rule ext)
  apply (simp add: prod-2-id-def prod-2-def)
  done

```

2.4.2 Combinators for Transition Policies

For constructing transition policies, two additional combinators are required: one combines state transitions by pairing the states, the other works equivalently on general maps.

```

definition parallel-map :: ( $'\alpha \rightarrow '\beta$ )  $\Rightarrow$  ( $'\delta \rightarrow '\gamma$ )  $\Rightarrow$ 
  ( $'\alpha \times '\delta \rightarrow '\beta \times '\gamma$ ) (infixr  $\langle \otimes_M \rangle$  60)
  where  $p1 \otimes_M p2 = (\lambda (x,y). \text{case } p1 \text{ } x \text{ of } [d1] \Rightarrow$ 
    ( $\text{case } p2 \text{ } y \text{ of } [d2] \Rightarrow [(d1,d2)]$ )
    |  $\perp \Rightarrow \perp$ )
    |  $\perp \Rightarrow \perp$ )

```

```

definition parallel-st :: ( $'i \times '\sigma \rightarrow '\sigma$ )  $\Rightarrow$  ( $'i \times '\sigma' \rightarrow '\sigma'$ )  $\Rightarrow$ 
  ( $'i \times '\sigma \times '\sigma' \rightarrow '\sigma \times '\sigma'$ ) (infixr  $\langle \otimes_S \rangle$  60)

```

where

```

 $p1 \otimes_S p2 = (p1 \otimes_M p2) \circ (\lambda (a,b,c). ((a,b),a,c))$ 

```

2.4.3 Range Splitting

The following combinator is a special case of both a parallel composition operator and a range splitting operator. Its primary use case is when combining a policy with state transitions.

```

definition comp-ran-split :: [ $'\alpha \rightarrow '\gamma$ ]  $\times$  ( $'\alpha \rightarrow '\gamma$ ),  $'d \mapsto '\beta$ ]  $\Rightarrow$  ( $'d \times '\alpha$ )  $\mapsto$  ( $'\beta \times '\gamma$ )
  (infixr  $\langle \otimes_{\nabla} \rangle$  100)
where  $P \otimes_{\nabla} p \equiv \lambda x. \text{case } p \text{ } (fst \ x) \text{ of}$ 
  [allow  $y$ ]  $\Rightarrow$  ( $\text{case } ((fst \ P) \ (snd \ x)) \text{ of } \perp \Rightarrow \perp \mid [z] \Rightarrow [allow$ 
  ( $y,z$ )]])
  | [deny  $y$ ]  $\Rightarrow$  ( $\text{case } ((snd \ P) \ (snd \ x)) \text{ of } \perp \Rightarrow \perp \mid [z] \Rightarrow [deny$ 
  ( $y,z$ )]])
  |  $\perp \Rightarrow \perp$ 

```

An alternative characterisation of the operator is as follows:

lemma *comp-ran-split-charn*:

```

 $(f, g) \otimes_{\nabla} p = ($ 

```

```

(((p ▷ Allow) ⊗∨A (Ap f)) ⊕
((p ▷ Deny) ⊗∨A (Dp g)))
apply (rule ext)
apply (simp add: comp-ran-split-def map-add-def o-def ran-restrict-def image-def
Allow-def Deny-def dom-restrict-def prod-orA-def
allow-pfun-def deny-pfun-def
split:option.splits decision.splits)
apply (auto)
done

```

2.4.4 Distributivity of the parallel combinators

```

lemma distr-or1-a: (F = F1 ⊕ F2) ⇒ (((N ⊗1 F) o f) =
(((N ⊗1 F1) o f) ⊕ ((N ⊗1 F2) o f)))
apply (rule ext)
apply (simp add: prod-1-def map-add-def
split: decision.splits option.splits)
subgoal for x
apply (case-tac f x)
apply (simp-all add: prod-1-def map-add-def
split: decision.splits option.splits)
done
done

```

```

lemma distr-or1: (F = F1 ⊕ F2) ⇒ ((g o-f ((N ⊗1 F) o f)) =
((g o-f ((N ⊗1 F1) o f)) ⊕ (g o-f ((N ⊗1 F2) o f))))
apply (rule ext)+
apply (simp add: prod-1-def map-add-def policy-range-comp-def
split: decision.splits option.splits)
subgoal for x
apply (case-tac f x)
apply (simp-all add: prod-1-def map-add-def
split: decision.splits option.splits)
done
done

```

```

lemma distr-or2-a: (F = F1 ⊕ F2) ⇒ (((N ⊗2 F) o f) =
(((N ⊗2 F1) o f) ⊕ ((N ⊗2 F2) o f)))
apply (rule ext)
apply (simp add: prod-2-id-def prod-2-def map-add-def
split: decision.splits option.splits)
subgoal for x
apply (case-tac f x)
apply (simp-all add: prod-2-def map-add-def

```

```

    split: decision.splits option.splits)
done
done

lemma distr-or2: (F = F1 ⊕ F2) ⇒ ((r o-f ((N ⊗2 F) o f)) =
    ((r o-f ((N ⊗2 F1) o f)) ⊕ (r o-f ((N ⊗2 F2) o f))))
apply (rule ext)
apply (simp add: prod-2-id-def prod-2-def map-add-def policy-range-comp-def
    split: decision.splits option.splits)
subgoal for x
  apply (case-tac f x)
  apply (simp-all add: prod-2-def map-add-def
    split: decision.splits option.splits)
done
done

lemma distr-orA: (F = F1 ⊕ F2) ⇒ ((g o-f ((N ⊗∨A F) o f)) =
    ((g o-f ((N ⊗∨A F1) o f)) ⊕ (g o-f ((N ⊗∨A F2) o f))))
apply (rule ext)+
apply (simp add: prod-orA-def map-add-def policy-range-comp-def
    split: decision.splits option.splits)
subgoal for x
  apply (case-tac f x)
  apply (simp-all add: map-add-def
    split: decision.splits option.splits)
done
done

lemma distr-orD: (F = F1 ⊕ F2) ⇒ ((g o-f ((N ⊗∨D F) o f)) =
    ((g o-f ((N ⊗∨D F1) o f)) ⊕ (g o-f ((N ⊗∨D F2) o f))))
apply (rule ext)+
apply (simp add: prod-orD-def map-add-def policy-range-comp-def
    split: decision.splits option.splits)
subgoal for x
  apply (case-tac f x)
  apply (simp-all add: map-add-def
    split: decision.splits option.splits)
done
done

lemma coerc-assoc: (r o-f P) o d = r o-f (P o d)
apply (simp add: policy-range-comp-def)
apply (rule ext)
apply (simp split: option.splits decision.splits)

```

```

done

lemmas ParallelDefs = prod-orA-def prod-orD-def prod-1-def prod-2-def paral-
lel-map-def
                        parallel-st-def comp-ran-split-def

end

```

2.5 Properties on Policies

```

theory
  Analysis
  imports
    ParallelComposition
    SeqComposition
begin

```

In this theory, several standard policy properties are paraphrased in UPF terms.

2.5.1 Basic Properties

A Policy Has no Gaps

```

definition gap-free :: ('a  $\mapsto$  'b)  $\Rightarrow$  bool
where gap-free p = (dom p = UNIV)

```

Comparing Policies

Policy p is more defined than q:

```

definition more-defined :: ('a  $\mapsto$  'b)  $\Rightarrow$  ('a  $\mapsto$  'b)  $\Rightarrow$  bool
where more-defined p q = (dom q  $\subseteq$  dom p)

```

```

definition strictly-more-defined :: ('a  $\mapsto$  'b)  $\Rightarrow$  ('a  $\mapsto$  'b)  $\Rightarrow$  bool
where strictly-more-defined p q = (dom q  $\subset$  dom p)

```

lemma *strictly-more-vs-more*: *strictly-more-defined p q \implies more-defined p q*

```

unfolding more-defined-def strictly-more-defined-def
by auto

```

Policy p is more permissive than q:

```

definition more-permissive :: ('a  $\mapsto$  'b)  $\Rightarrow$  ('a  $\mapsto$  'b)  $\Rightarrow$  bool (infixl  $\sqsubseteq_A$  60)
where p  $\sqsubseteq_A$  q = ( $\forall$  x. (case q x of [allow y]  $\Rightarrow$  ( $\exists$  z. (p x = [allow z]))
    | [deny y]  $\Rightarrow$  True
    |  $\perp$   $\Rightarrow$  True))

```

lemma *more-permissive-refl* : $p \sqsubseteq_A p$
unfolding *more-permissive-def*
by(*auto split : option.split decision.split*)

lemma *more-permissive-trans* : $p \sqsubseteq_A p' \implies p' \sqsubseteq_A p'' \implies p \sqsubseteq_A p''$
unfolding *more-permissive-def*
apply(*auto split : option.split decision.split*)
subgoal for $x\ y$
apply(*erule-tac x = x and*
 $P = \lambda x. \text{case } p''\ x \text{ of } \perp \Rightarrow \text{True}$
 $\quad | \text{[allow } y] \Rightarrow \exists z. p'\ x = \text{[allow } z]$
 $\quad | \text{[deny } y] \Rightarrow \text{True in allE}$)
apply(*simp, elim exE*)
by(*erule-tac x = x in allE, simp*)
done

Policy p is more rejective than q :

definition *more-rejective* :: $('a \mapsto 'b) \Rightarrow ('a \mapsto 'b) \Rightarrow \text{bool}$ (**infixl** $\langle \sqsubseteq_D \rangle$ 60)
where $p \sqsubseteq_D q = (\forall x. (\text{case } q\ x \text{ of } \text{[deny } y] \Rightarrow (\exists z. (p\ x = \text{[deny } z]))$
 $\quad | \text{[allow } y] \Rightarrow \text{True}$
 $\quad | \perp \Rightarrow \text{True}))$

lemma *more-rejective-trans* : $p \sqsubseteq_D p' \implies p' \sqsubseteq_D p'' \implies p \sqsubseteq_D p''$
unfolding *more-rejective-def*
apply(*auto split : option.split decision.split*)
subgoal for $x\ y$
apply(*erule-tac x = x and*
 $P = \lambda x. \text{case } p''\ x \text{ of } \perp \Rightarrow \text{True}$
 $\quad | \text{[allow } y] \Rightarrow \text{True}$
 $\quad | \text{[deny } y] \Rightarrow \exists z. p'\ x = \text{[deny } z] \text{ in allE}$)
apply(*simp, elim exE*)
by(*erule-tac x = x in allE, simp*)
done

lemma *more-rejective-refl* : $p \sqsubseteq_D p$
unfolding *more-rejective-def*
by(*auto split : option.split decision.split*)

lemma $A_f\ f \sqsubseteq_A p$

unfolding *more-permissive-def allow-all-fun-def allow-pfun-def*
by(*auto split: option.split decision.split*)

lemma $A_I \sqsubseteq_A p$

unfolding *more-permissive-def allow-all-fun-def allow-pfun-def allow-all-id-def*
by(*auto split: option.split decision.split*)

2.5.2 Combined Data-Policy Refinement

definition *policy-refinement* ::

$(\text{'a} \mapsto \text{'b}) \Rightarrow (\text{'a}' \Rightarrow \text{'a}) \Rightarrow (\text{'b}' \Rightarrow \text{'b}) \Rightarrow (\text{'a}' \mapsto \text{'b'}) \Rightarrow \text{bool}$

$(\leftarrow \sqsubseteq_{-, -} \rightarrow [50, 50, 50, 50] 50)$

where $p \sqsubseteq_{\text{abs}_a, \text{abs}_b} q \equiv$

$(\forall a. \text{case } p \text{ a of}$

$\perp \Rightarrow \text{True}$

$| \text{[allow } y] \Rightarrow (\forall a' \in \{x. \text{abs}_a \ x = a\}.$
 $\exists b'. q \ a' = \text{[allow } b']$
 $\wedge \text{abs}_b \ b' = y)$

$| \text{[deny } y] \Rightarrow (\forall a' \in \{x. \text{abs}_a \ x = a\}.$
 $\exists b'. q \ a' = \text{[deny } b']$
 $\wedge \text{abs}_b \ b' = y))$

theorem *polref-refl*: $p \sqsubseteq_{\text{id}, \text{id}} p$

unfolding *policy-refinement-def*

by(*auto split: option.split decision.split*)

theorem *polref-trans*:

assumes $A: p \sqsubseteq_{f, g} p'$

and $B: p' \sqsubseteq_{f', g'} p''$

shows $p \sqsubseteq_{f \circ f', g \circ g'} p''$

apply(*insert A B*)

unfolding *policy-refinement-def*

apply(*auto split: option.split decision.split simp: o-def*)[1]

subgoal for $a \ a'$

apply(*erule-tac x=f (f' a') in allE, simp*)[1]

apply(*erule-tac x=f' a' in allE, auto*)[1]

apply(*erule-tac x=(f' a') in allE, auto*)[1]

done

subgoal for $a \ a'$

apply(*erule-tac x=f (f' a') in allE, simp*)[1]

apply(*erule-tac x=f' a' in allE, auto*)[1]

apply(*erule-tac x=(f' a') in allE, auto*)[1]

done

done

2.5.3 Equivalence of Policies

Equivalence over domain D

definition $p\text{-eq-dom} :: ('a \mapsto 'b) \Rightarrow 'a \text{ set} \Rightarrow ('a \mapsto 'b) \Rightarrow \text{bool}$ ($\langle _ \approx _ \rangle$ \rightarrow $[60,60,60]60$)
where $p \approx_D q = (\forall x \in D. p\ x = q\ x)$

p and q have no conflicts

definition $\text{no-conflicts} :: ('a \mapsto 'b) \Rightarrow ('a \mapsto 'b) \Rightarrow \text{bool}$ **where**
 $\text{no-conflicts } p\ q = (\text{dom } p = \text{dom } q \wedge (\forall x \in (\text{dom } p).$
 $(\text{case } p\ x\ \text{of } \lfloor \text{allow } y \rfloor \Rightarrow (\exists z. q\ x = \lfloor \text{allow } z \rfloor)$
 $\mid \lfloor \text{deny } y \rfloor \Rightarrow (\exists z. q\ x = \lfloor \text{deny } z \rfloor))))$

lemma policy-eq :

assumes $p\text{-over-}qA$: $p \sqsubseteq_A q$
and $q\text{-over-}pA$: $q \sqsubseteq_A p$
and $p\text{-over-}qD$: $q \sqsubseteq_D p$
and $q\text{-over-}pD$: $p \sqsubseteq_D q$
and dom-eq : $\text{dom } p = \text{dom } q$
shows $\text{no-conflicts } p\ q$
apply ($\text{insert } p\text{-over-}qA\ q\text{-over-}pA\ p\text{-over-}qD\ q\text{-over-}pD\ \text{dom-eq}$)
apply ($\text{simp add: no-conflicts-def more-permissive-def more-rejective-def}$
 $\text{split: option.splits decision.splits}$)
apply (safe)
apply ($\text{metis domI domIff dom-eq}$)
apply (metis)
done

Miscellaneous

lemma dom-inter : $\llbracket \text{dom } p \cap \text{dom } q = \{\}; p\ x = \lfloor y \rfloor \rrbracket \Longrightarrow q\ x = \perp$
by (auto)

lemma dom-eq : $\text{dom } p \cap \text{dom } q = \{\} \Longrightarrow p \oplus_A q = p \oplus_D q$
unfolding $\text{override-A-def override-D-def}$
by ($\text{rule ext, auto simp: dom-def split: prod.splits option.splits decision.splits}$)

lemma dom-split-alt-def : $(f, g) \Delta p = (\text{dom}(p \triangleright \text{Allow}) \triangleleft (A_f f)) \oplus (\text{dom}(p \triangleright \text{Deny})$
 $\triangleleft (D_f g))$
apply (rule ext)
apply ($\text{simp add: dom-split2-def Allow-def Deny-def dom-restrict-def}$
 $\text{deny-all-fun-def allow-all-fun-def map-add-def}$)
apply ($\text{simp split: option.splits decision.splits}$)
apply ($\text{auto simp: map-add-def o-def deny-pfun-def ran-restrict-def image-def}$)
done

end

2.6 Policy Transformations

theory

Normalisation

imports

SeqComposition

ParallelComposition

begin

This theory provides the formalisations required for the transformation of UPF policies. A typical usage scenario can be observed in the firewall case study [12].

2.6.1 Elementary Operators

We start by providing several operators and theorems useful when reasoning about a list of rules which should eventually be interpreted as combined using the standard override operator.

The following definition takes as argument a list of rules and returns a policy where the rules are combined using the standard override operator.

definition *list2policy*::('a \mapsto 'b) list \Rightarrow ('a \mapsto 'b) **where**

list2policy l = foldr (λ x y. (x \oplus y)) l \emptyset

Determine the position of element of a list.

fun *position* :: 'a \Rightarrow 'a list \Rightarrow nat **where**

position a [] = 0

|(*position* a (x#xs)) = (if a = x then 1 else (Suc (*position* a xs)))

Provides the first applied rule of a policy given as a list of rules.

fun *applied-rule* **where**

applied-rule C a (x#xs) = (if a \in dom (C x) then (Some x)
else (*applied-rule* C a xs))

|*applied-rule* C a [] = None

The following is used if the list is constructed backwards.

definition *applied-rule-rev* **where**

applied-rule-rev C a x = *applied-rule* C a (rev x)

The following is a typical policy transformation. It can be applied to any type of policy and removes all the rules from a policy with an empty domain. It takes two arguments: a semantic interpretation function and a list of rules.

fun *rm-MT-rules* **where**

```

rm-MT-rules C (x#xs) = (if dom (C x) = {}
                        then rm-MT-rules C xs
                        else x#(rm-MT-rules C xs))
|rm-MT-rules C [] = []

```

The following invariant establishes that there are no rules with an empty domain in a list of rules.

```

fun none-MT-rules where
  none-MT-rules C (x#xs) = (dom (C x) ≠ {}) ∧ (none-MT-rules C xs)
|none-MT-rules C [] = True

```

The following related invariant establishes that the policy has not a completely empty domain.

```

fun not-MT where
  not-MT C (x#xs) = (if (dom (C x) = {}) then (not-MT C xs) else True)
|not-MT C [] = False

```

Next, a few theorems about the two invariants and the transformation:

```

lemma none-MT-rules-vs-notMT: none-MT-rules C p ⇒ p ≠ [] ⇒ not-MT C p
apply (induct p)
apply (simp-all)
done

```

```

lemma rmnMT: none-MT-rules C (rm-MT-rules C p)
apply (induct p)
apply (simp-all)
done

```

```

lemma rmnMT2: none-MT-rules C p ⇒ (rm-MT-rules C p) = p
apply (induct p)
apply (simp-all)
done

```

```

lemma nMTcharn: none-MT-rules C p = (∀ r ∈ set p. dom (C r) ≠ {})
apply (induct p)
apply (simp-all)
done

```

```

lemma nMTeqSet: set p = set s ⇒ none-MT-rules C p = none-MT-rules C s
apply (simp add: nMTcharn)
done

```

```

lemma notMTnMT: [a ∈ set p; none-MT-rules C p] ⇒ dom (C a) ≠ {}
apply (simp add: nMTcharn)

```

done

lemma *none-MT-rulesconc*: *none-MT-rules C (a@[b]) \implies none-MT-rules C a*

apply (*induct a*)
apply (*simp-all*)
done

lemma *nMTtail*: *none-MT-rules C p \implies none-MT-rules C (tl p)*

apply (*induct p*)
apply (*simp-all*)
done

lemma *not-MTimpnotMT[simp]*: *not-MT C p \implies p \neq []*

apply (*auto*)
done

lemma *SR3nMT*: \neg *not-MT C p \implies rm-MT-rules C p = []*

apply (*induct p*)
apply (*auto simp: if-splits*)
done

lemma *NMPcharn*: $\llbracket a \in \text{set } p; \text{dom } (C a) \neq \{\} \rrbracket \implies \text{not-MT } C p$

apply (*induct p*)
apply (*auto simp: if-splits*)
done

lemma *NMPrm*: *not-MT C p \implies not-MT C (rm-MT-rules C p)*

apply (*induct p*)
apply (*simp-all*)
done

Next, a few theorems about `applied_rule`:

lemma *mrconc*: *applied-rule-rev C x p = Some a \implies applied-rule-rev C x (b#p) = Some a*

proof (*induct p rule: rev-induct*)
case *Nil* **show** *?case* **using** *Nil*
by (*simp add: applied-rule-rev-def*)
next
case (*snoc xs x*) **show** *?case* **using** *snoc*
apply (*simp add: applied-rule-rev-def if-splits*)
by (*metis option.inject*)
qed

lemma *mreq-end*: $\llbracket \text{applied-rule-rev } C x b = \text{Some } r; \text{applied-rule-rev } C x c = \text{Some } r \rrbracket$

\implies
 $applied\text{-rule}\text{-rev } C \ x \ (a\#b) = applied\text{-rule}\text{-rev } C \ x \ (a\#c)$
by (*simp add: mrconc*)

lemma *mrconcNone*: $applied\text{-rule}\text{-rev } C \ x \ p = None \implies$
 $applied\text{-rule}\text{-rev } C \ x \ (b\#p) = applied\text{-rule}\text{-rev } C \ x \ [b]$

proof (*induct p rule: rev-induct*)
case *Nil* **show** *?case*
by (*simp add: applied-rule-rev-def*)
next
case (*snoc ys y*) **show** *?case* **using** *snoc*
proof (*cases x ∈ dom (C ys)*)
case *True* **show** *?thesis* **using** *True snoc*
by (*auto simp: applied-rule-rev-def*)
next
case *False* **show** *?thesis* **using** *False snoc*
by (*auto simp: applied-rule-rev-def*)
qed
qed

lemma *mreq-endNone*: $\llbracket applied\text{-rule}\text{-rev } C \ x \ b = None; applied\text{-rule}\text{-rev } C \ x \ c = None \rrbracket$
 \implies
 $applied\text{-rule}\text{-rev } C \ x \ (a\#b) = applied\text{-rule}\text{-rev } C \ x \ (a\#c)$
by (*metis mrconcNone*)

lemma *mreq-end2*: $applied\text{-rule}\text{-rev } C \ x \ b = applied\text{-rule}\text{-rev } C \ x \ c \implies$
 $applied\text{-rule}\text{-rev } C \ x \ (a\#b) = applied\text{-rule}\text{-rev } C \ x \ (a\#c)$
apply (*case-tac applied-rule-rev C x b = None*)
apply (*auto intro: mreq-end mreq-endNone*)
done

lemma *mreq-end3*: $applied\text{-rule}\text{-rev } C \ x \ p \neq None \implies$
 $applied\text{-rule}\text{-rev } C \ x \ (b \# p) = applied\text{-rule}\text{-rev } C \ x \ (p)$
by (*auto simp: mrconc*)

lemma *mrNoneMT*: $\llbracket r \in set \ p; applied\text{-rule}\text{-rev } C \ x \ p = None \rrbracket \implies$
 $x \notin dom \ (C \ r)$
proof (*induct p rule: rev-induct*)
case *Nil* **show** *?case* **using** *Nil*
by (*simp add: applied-rule-rev-def*)
next
case (*snoc y ys*) **show** *?case* **using** *snoc*
proof (*cases r ∈ set ys*)
case *True* **show** *?thesis* **using** *snoc True*

```

    by (simp add: applied-rule-rev-def split: if-split-asm)
next
case False show ?thesis using snoc False
    by (simp add: applied-rule-rev-def split: if-split-asm)
qed
qed

```

2.6.2 Distributivity of the Transformation.

The scenario is the following (can be applied iteratively):

- Two policies are combined using one of the parallel combinators
- (e.g. $P = P1 P2$) (At least) one of the constituent policies has
- a normalisation procedures, which as output produces a list of
- policies that are semantically equivalent to the original policy if
- combined from left to right using the override operator.

The following function is crucial for the distribution. Its arguments are a policy, a list of policies, a parallel combinator, and a range and a domain coercion function.

```

fun prod-list :: ('α ↦ 'β) ⇒ (('γ ↦ 'δ) list) ⇒
  (('α ↦ 'β) ⇒ ('γ ↦ 'δ) ⇒ (('α × 'γ) ↦ ('β × 'δ))) ⇒
  (('β × 'δ) ⇒ 'y) ⇒ ('x ⇒ ('α × 'γ)) ⇒
  (('x ↦ 'y) list) (infixr <⊗L> 54) where
  prod-list x (y#ys) par-comb ran-adapt dom-adapt =
    ((ran-adapt o-f ((par-comb x y) o dom-adapt))#(prod-list x ys par-comb ran-adapt
  dom-adapt))
| prod-list x [] par-comb ran-adapt dom-adapt = []

```

An instance, as usual there are four of them.

```

definition prod-2-list :: [('α ↦ 'β), (('γ ↦ 'δ) list)] ⇒
  (('β × 'δ) ⇒ 'y) ⇒ ('x ⇒ ('α × 'γ)) ⇒
  (('x ↦ 'y) list) (infixr <⊗2L> 55) where
  x ⊗2L y = (λ d r. (x ⊗L y) (⊗2) d r)

```

```

lemma list2listNMT: x ≠ [] ⇒ map sem x ≠ []
apply (case-tac x)
apply (simp-all)
done

```

```

lemma two-conc: (prod-list x (y#ys) p r d) = ((r o-f ((p x y) o d))#(prod-list x ys p
  r d))
by simp

```

The following two invariants establish if the law of distributivity holds for a combinator and if an operator is strict regarding undefinedness.

definition *is-distr* **where**

$$\text{is-distr } p = (\lambda g f. (\forall N P1 P2. ((g \text{ o-f } ((p N (P1 \oplus P2)) \text{ o-f})) = ((g \text{ o-f } ((p N P1) \text{ o-f})) \oplus (g \text{ o-f } ((p N P2) \text{ o-f}))))))$$

definition *is-strict* **where**

$$\text{is-strict } p = (\lambda r d. \forall P1. (r \text{ o-f } (p P1 \emptyset \circ d)) = \emptyset)$$

lemma *is-distr-orD*: *is-distr* $(\otimes_{\vee D})$ *d r*

apply (*simp add: is-distr-def*)

apply (*rule allI*)⁺

apply (*rule distr-orD*)

apply (*simp*)

done

lemma *is-strict-orD*: *is-strict* $(\otimes_{\vee D})$ *d r*

apply (*simp add: is-strict-def*)

apply (*simp add: policy-range-comp-def*)

done

lemma *is-distr-2*: *is-distr* (\otimes_2) *d r*

apply (*simp add: is-distr-def*)

apply (*rule allI*)⁺

apply (*rule distr-or2*)

by *simp*

lemma *is-strict-2*: *is-strict* (\otimes_2) *d r*

apply (*simp only: is-strict-def*)

apply *simp*

apply (*simp add: policy-range-comp-def*)

done

lemma *domStart*: $t \in \text{dom } p1 \implies (p1 \oplus p2) t = p1 t$

apply (*simp add: map-add-dom-app-simps*)

done

lemma *notDom*: $x \in \text{dom } A \implies \neg A x = \text{None}$

apply *auto*

done

The following theorems are crucial: they establish the correctness of the distribution.

lemma *Norm-Distr-1*: $((r \text{ o-f } (((\otimes_1) P1 (\text{list2policy } P2)) \text{ o-f})) x = ((\text{list2policy } ((P1 \otimes_L P2) (\otimes_1) r d)) x))$

```

proof (induct P2)
  case Nil show ?case
    by (simp add: policy-range-comp-def list2policy-def)
next
  case (Cons p ps) show ?case using Cons
  proof (cases x ∈ dom (r o-f ((P1 ⊗1 p) ∘ d)))
    case True show ?thesis using True
      by (auto simp: list2policy-def policy-range-comp-def prod-1-def
        split: option.splits decision.splits prod.splits)
    next
      case False show ?thesis using Cons False
        by (auto simp: list2policy-def policy-range-comp-def map-add-dom-app-simps(3)
          prod-1-def
            split: option.splits decision.splits prod.splits)
  qed
qed

```

```

lemma Norm-Distr-2: ((r o-f (((⊗2) P1 (list2policy P2)) ∘ d)) x =
  ((list2policy ((P1 ⊗L P2) (⊗2) r d)) x)) proof (induct P2)
  case Nil show ?case
    by (simp add: policy-range-comp-def list2policy-def)
next
  case (Cons p ps) show ?case using Cons
  proof (cases x ∈ dom (r o-f ((P1 ⊗2 p) ∘ d)))
    case True show ?thesis using True
      by (auto simp: list2policy-def prod-2-def policy-range-comp-def
        split: option.splits decision.splits prod.splits)
    next
      case False show ?thesis using Cons False
        by (auto simp: policy-range-comp-def list2policy-def map-add-dom-app-simps(3)
          prod-2-def
            split: option.splits decision.splits prod.splits)
  qed
qed

```

```

lemma Norm-Distr-A: ((r o-f (((⊗∨A) P1 (list2policy P2)) ∘ d)) x =
  ((list2policy ((P1 ⊗L P2) (⊗∨A) r d)) x))
proof (induct P2)
  case Nil show ?case
    by (simp add: policy-range-comp-def list2policy-def)
next
  case (Cons p ps) show ?case using Cons
  proof (cases x ∈ dom (r o-f ((P1 ⊗∨A p) ∘ d)))
    case True show ?thesis using True

```

```

    by (auto simp: policy-range-comp-def list2policy-def prod-orA-def
        split: option.splits decision.splits prod.splits)
  next
    case False show ?thesis using Cons False
      by (auto simp: policy-range-comp-def list2policy-def map-add-dom-app-simps(3)
          prod-orA-def
              split: option.splits decision.splits prod.splits)
  qed
qed

```

lemma *Norm-Distr-D*: $((r \text{ o-f } (((\otimes_{\vee D}) P1 \text{ (list2policy } P2)) \text{ o } d)) \text{ x} =$
 $((\text{list2policy } ((P1 \otimes_L P2) (\otimes_{\vee D}) r \text{ d})) \text{ x}))$

```

proof (induct P2)
  case Nil show ?case
    by (simp add: policy-range-comp-def list2policy-def)
  next
    case (Cons p ps) show ?case using Cons
  proof (cases  $x \in \text{dom } (r \text{ o-f } ((P1 \otimes_{\vee D}) p) \text{ o } d))$ )
    case True show ?thesis using True
      by (auto simp: policy-range-comp-def list2policy-def prod-orD-def
          split: option.splits decision.splits prod.splits)
    next
      case False show ?thesis using Cons False
        by (auto simp: policy-range-comp-def list2policy-def map-add-dom-app-simps(3)
            prod-orD-def
                split: option.splits decision.splits prod.splits)
  qed
qed

```

Some domain reasoning

lemma *domSubsetDistr1*: $\text{dom } A = \text{UNIV} \implies \text{dom } ((\lambda(x, y). x) \text{ o-f } (A \otimes_1 B) \text{ o } (\lambda$
 $x. (x, x))) = \text{dom } B$

```

apply (rule set-eqI)
apply (rule iffI)
apply (auto simp: prod-1-def policy-range-comp-def dom-def
    split: decision.splits option.splits prod.splits)
done

```

lemma *domSubsetDistr2*: $\text{dom } A = \text{UNIV} \implies \text{dom } ((\lambda(x, y). x) \text{ o-f } (A \otimes_2 B) \text{ o } (\lambda$
 $x. (x, x))) = \text{dom } B$

```

apply (rule set-eqI)
apply (rule iffI)
apply (auto simp: prod-2-def policy-range-comp-def dom-def)

```

```

    split: decision.splits option.splits prod.splits)
done

lemma domSubsetDistrA: dom A = UNIV  $\implies$  dom (( $\lambda(x, y). x$ ) o-f ( $A \otimes_{\vee A} B$ ) o
( $\lambda x. (x, x)$ )) = dom B
  apply (rule set-eqI)
  apply (rule iffI)
  apply (auto simp: prod-orA-def policy-range-comp-def dom-def
    split: decision.splits option.splits prod.splits)
done

lemma domSubsetDistrD: dom A = UNIV  $\implies$  dom (( $\lambda(x, y). x$ ) o-f ( $A \otimes_{\vee D} B$ ) o
( $\lambda x. (x, x)$ )) = dom B
  apply (rule set-eqI)
  apply (rule iffI)
  apply (auto simp: prod-orD-def policy-range-comp-def dom-def
    split: decision.splits option.splits prod.splits)
done
end

```

2.7 Policy Transformation for Testing

theory

NormalisationTestSpecification

imports

Normalisation

begin

This theory provides functions and theorems which are useful if one wants to test policy which are transformed. Most exist in two versions: one where the domains of the rules of the list (which is the result of a transformation) are pairwise disjoint, and one where this applies not for the last rule in a list (which is usually a default rules).

The examples in the firewall case study provide a good documentation how these theories can be applied.

This invariant establishes that the domains of a list of rules are pairwise disjoint.

fun *disjDom* **where**

disjDom ($x\#xs$) = ($(\forall y \in (\text{set } xs). \text{dom } x \cap \text{dom } y = \{\}) \wedge \text{disjDom } xs$)
 $|\text{disjDom } [] = \text{True}$

fun *PUTList* :: ($'a \mapsto 'b$) \Rightarrow $'a \Rightarrow ('a \mapsto 'b)$ list \Rightarrow bool

where

PUTList *PUT* x ($p\#ps$) = ($(x \in \text{dom } p \longrightarrow (\text{PUT } x = p \ x)) \wedge (\text{PUTList } \text{PUT } x \ ps)$)
 $|\text{PUTList } \text{PUT } x \ [] = \text{True}$

lemma *distrPUTL1*: $x \in \text{dom } P \implies (\text{list2policy } PL) x = P x$
 $\implies (\text{PUTList } PUT x PL \implies (PUT x = P x))$

apply (*induct PL*)
apply (*auto simp: list2policy-def dom-def*)
done

lemma *PUTList-None*: $x \notin \text{dom } (\text{list2policy } list) \implies \text{PUTList } PUT x list$

apply (*induct list*)
apply (*auto simp: list2policy-def dom-def*)
done

lemma *PUTList-DomMT*:

$(\forall y \in \text{set } list. \text{dom } a \cap \text{dom } y = \{\}) \implies x \in (\text{dom } a) \implies x \notin \text{dom } (\text{list2policy } list)$

apply (*induct list*)
apply (*auto simp: dom-def list2policy-def*)
done

lemma *distrPUTL2*:

$x \in \text{dom } P \implies (\text{list2policy } PL) x = P x \implies \text{disjDom } PL \implies (PUT x = P x) \implies$
 $\text{PUTList } PUT x PL$

apply (*induct PL*)
apply (*simp-all add: list2policy-def*)
apply (*auto*)[1]
subgoal for a PL p
apply (*case-tac x \in dom a*)
apply (*case-tac list2policy PL x = P x*)
apply (*simp add: list2policy-def*)
apply (*rule PUTList-None*)
apply (*rule-tac a = a in PUTList-DomMT*)
apply (*simp-all add: list2policy-def dom-def*)
done
done

lemma *distrPUTL*:

$\llbracket x \in \text{dom } P; (\text{list2policy } PL) x = P x; \text{disjDom } PL \rrbracket \implies (PUT x = P x) = \text{PUTList } PUT x PL$

apply (*rule iffI*)
apply (*rule distrPUTL2*)
apply (*simp-all*)
apply (*rule-tac PL = PL in distrPUTL1*)
apply (*auto*)
done

It makes sense to cater for the common special case where the normalisation returns a list where the last element is a default-catch-all rule. It seems easier to cater for this globally, rather than to require the normalisation procedures to do this.

```
fun gatherDomain-aux where
  gatherDomain-aux (x#xs) = (dom x ∪ (gatherDomain-aux xs))
|gatherDomain-aux [] = {}
```

```
definition gatherDomain where gatherDomain p = (gatherDomain-aux (butlast p))
```

```
definition PUTListGD where PUTListGD PUT x p =
  (((x ∉ (gatherDomain p) ∧ x ∈ dom (last p)) → PUT x = (last p) x) ∧
   (PUTList PUT x (butlast p)))
```

```
definition disjDomGD where disjDomGD p = disjDom (butlast p)
```

```
lemma distrPUTLG1:  $\llbracket x \in \text{dom } P; (\text{list2policy } PL) x = P x; \text{PUTListGD } \text{PUT } x \text{ } PL \rrbracket$ 
 $\implies \text{PUT } x = P x$ 
```

```
  apply (induct PL)
  apply (simp-all add: domIff PUTListGD-def disjDomGD-def gatherDomain-def
list2policy-def)
  apply (auto simp: dom-def domIff split: if-split-asm)
done
```

```
lemma distrPUTLG2:
```

```
   $PL \neq [] \implies x \in \text{dom } P \implies (\text{list2policy } (PL)) x = P x \implies \text{disjDomGD } PL \implies$ 
   $(\text{PUT } x = P x) \implies \text{PUTListGD } \text{PUT } x \text{ } (PL)$ 
  apply (simp add: PUTListGD-def disjDomGD-def gatherDomain-def list2policy-def)
  apply (induct PL)
  apply (auto)
  apply (metis PUTList-DomMT PUTList-None domI)
done
```

```
lemma distrPUTLG:
```

```
   $\llbracket x \in \text{dom } P; (\text{list2policy } PL) x = P x; \text{disjDomGD } PL; PL \neq [] \rrbracket \implies$ 
   $(\text{PUT } x = P x) = \text{PUTListGD } \text{PUT } x \text{ } PL$ 
  apply (rule iffI)
  apply (rule distrPUTLG2)
  apply (simp-all)
  apply (rule-tac PL = PL in distrPUTLG1)
  apply (auto)
done
```

end

2.8 Putting Everything Together: UPF

theory

UPF

imports

Normalisation

NormalisationTestSpecification

Analysis

begin

This is the top-level theory for the Unified Policy Framework (UPF) and, thus, builds the base theory for using UPF. For the moment, we only define a set of lemmas for all core UPF definitions that is useful for using UPF:

lemmas *UPFDefs = UPFCoreDefs ParallelDefs ElementaryPoliciesDefs*

end

3 Example

In this chapter, we present a small example application of UPF for modeling access control for a Web service that might be used in a hospital. This scenario is motivated by our formalization of the NHS system [10, 13].

UPF was also successfully used for modeling network security policies such as the ones enforced by firewalls [12, 13]. These models were also used for generating test cases using HOL-TestGen [9].

3.1 Secure Service Specification

```
theory  
  Service  
  imports  
    UPF  
begin
```

In this section, we model a simple Web service and its access control model that allows the staff in a hospital to access health care records of patients.

3.1.1 Datatypes for Modelling Users and Roles

Users

First, we introduce a type for users that we use to model that each staff member has a unique id:

```
type-synonym user = int
```

Similarly, each patient has a unique id:

```
type-synonym patient = int
```

Roles and Relationships

In our example, we assume three different roles for members of the clinical staff:

```
datatype role = ClinicalPractitioner | Nurse | Clerical
```

We model treatment relationships (legitimate relationships) between staff and patients (respectively, their health records. This access control model is inspired by our detailed NHS model.

type-synonym $lr-id = int$
type-synonym $LR = lr-id \rightarrow (user\ set)$

The security context stores all the existing LRs.

type-synonym $\Sigma = patient \rightarrow LR$

The user context stores the roles the users are in.

type-synonym $v = user \rightarrow role$

3.1.2 Modelling Health Records and the Web Service API

Health Records

The content and the status of the entries of a health record

datatype $data = dummyContent$
datatype $status = Open \mid Closed$
type-synonym $entry-id = int$
type-synonym $entry = status \times user \times data$
type-synonym $SCR = (entry-id \rightarrow entry)$
type-synonym $DB = patient \rightarrow SCR$

The Web Service API

The operations provided by the service:

datatype $Operation = createSCR\ user\ role\ patient$
| $appendEntry\ user\ role\ patient\ entry-id\ entry$
| $deleteEntry\ user\ role\ patient\ entry-id$
| $readEntry\ user\ role\ patient\ entry-id$
| $readSCR\ user\ role\ patient$
| $addLR\ user\ role\ patient\ lr-id\ (user\ set)$
| $removeLR\ user\ role\ patient\ lr-id$
| $changeStatus\ user\ role\ patient\ entry-id\ status$
| $deleteSCR\ user\ role\ patient$
| $editEntry\ user\ role\ patient\ entry-id\ entry$

fun $is-createSCR$ **where**
| $is-createSCR\ (createSCR\ u\ r\ p) = True$
| $is-createSCR\ x = False$

fun $is-appendEntry$ **where**
| $is-appendEntry\ (appendEntry\ u\ r\ p\ e\ ei) = True$
| $is-appendEntry\ x = False$

fun $is-deleteEntry$ **where**

is-deleteEntry (*deleteEntry* *u r p e-id*) = *True*
| *is-deleteEntry* *x* = *False*

fun *is-readEntry* **where**
is-readEntry (*readEntry* *u r p e*) = *True*
| *is-readEntry* *x* = *False*

fun *is-readSCR* **where**
is-readSCR (*readSCR* *u r p*) = *True*
| *is-readSCR* *x* = *False*

fun *is-changeStatus* **where**
is-changeStatus (*changeStatus* *u r p s ei*) = *True*
| *is-changeStatus* *x* = *False*

fun *is-deleteSCR* **where**
is-deleteSCR (*deleteSCR* *u r p*) = *True*
| *is-deleteSCR* *x* = *False*

fun *is-addLR* **where**
is-addLR (*addLR* *u r lrid lr us*) = *True*
| *is-addLR* *x* = *False*

fun *is-removeLR* **where**
is-removeLR (*removeLR* *u r p lr*) = *True*
| *is-removeLR* *x* = *False*

fun *is-editEntry* **where**
is-editEntry (*editEntry* *u r p e-id s*) = *True*
| *is-editEntry* *x* = *False*

fun *SCR**Op* :: (*Operation* × *DB*) → *SCR* **where**
*SCR**Op* ((*createSCR* *u r p*), *S*) = *S p*
| *SCR**Op* ((*appendEntry* *u r p ei e*), *S*) = *S p*
| *SCR**Op* ((*deleteEntry* *u r p e-id*), *S*) = *S p*
| *SCR**Op* ((*readEntry* *u r p e*), *S*) = *S p*
| *SCR**Op* ((*readSCR* *u r p*), *S*) = *S p*
| *SCR**Op* ((*changeStatus* *u r p s ei*), *S*) = *S p*
| *SCR**Op* ((*deleteSCR* *u r p*), *S*) = *S p*
| *SCR**Op* ((*editEntry* *u r p e-id s*), *S*) = *S p*
| *SCR**Op* *x* = ⊥

fun *patientOfOp* :: *Operation* ⇒ *patient* **where**
patientOfOp (*createSCR* *u r p*) = *p*

```

|patientOfOp (appendEntry u r p e ei) = p
|patientOfOp (deleteEntry u r p e-id) = p
|patientOfOp (readEntry u r p e) = p
|patientOfOp (readSCR u r p) = p
|patientOfOp (changeStatus u r p s ei) = p
|patientOfOp (deleteSCR u r p) = p
|patientOfOp (addLR u r p lr ei) = p
|patientOfOp (removeLR u r p lr) = p
|patientOfOp (editEntry u r p e-id s) = p

```

```

fun userOfOp :: Operation ⇒ user where
  userOfOp (createSCR u r p) = u
|userOfOp (appendEntry u r p e ei) = u
|userOfOp (deleteEntry u r p e-id) = u
|userOfOp (readEntry u r p e) = u
|userOfOp (readSCR u r p) = u
|userOfOp (changeStatus u r p s ei) = u
|userOfOp (deleteSCR u r p) = u
|userOfOp (editEntry u r p e-id s) = u
|userOfOp (addLR u r p lr ei) = u
|userOfOp (removeLR u r p lr) = u

```

```

fun roleOfOp :: Operation ⇒ role where
  roleOfOp (createSCR u r p) = r
|roleOfOp (appendEntry u r p e ei) = r
|roleOfOp (deleteEntry u r p e-id) = r
|roleOfOp (readEntry u r p e) = r
|roleOfOp (readSCR u r p) = r
|roleOfOp (changeStatus u r p s ei) = r
|roleOfOp (deleteSCR u r p) = r
|roleOfOp (editEntry u r p e-id s) = r
|roleOfOp (addLR u r p lr ei) = r
|roleOfOp (removeLR u r p lr) = r

```

```

fun contentOfOp :: Operation ⇒ data where
  contentOfOp (appendEntry u r p ei e) = (snd (snd e))
|contentOfOp (editEntry u r p ei e) = (snd (snd e))

```

```

fun contentStatic :: Operation ⇒ bool where
  contentStatic (appendEntry u r p ei e) = ((snd (snd e)) = dummyContent)
|contentStatic (editEntry u r p ei e) = ((snd (snd e)) = dummyContent)
|contentStatic x = True

```

```

fun allContentStatic where

```

$$\begin{aligned} allContentStatic (x\#xs) &= (contentStatic x \wedge allContentStatic xs) \\ allContentStatic [] &= True \end{aligned}$$

3.1.3 Modelling Access Control

In the following, we define a rather complex access control model for our scenario that extends traditional role-based access control (RBAC) [20] with treatment relationships and sealed envelopes. Sealed envelopes (see [13] for details) are a variant of break-the-glass access control (see [8] for a general motivation and explanation of break-the-glass access control).

Sealed Envelopes

type-synonym $SEPolicy = (Operation \times DB \mapsto unit)$

definition $getEntry :: DB \Rightarrow patient \Rightarrow entry-id \Rightarrow entry\ option$ **where**
 $getEntry\ S\ p\ e-id = (case\ S\ p\ of\ \perp \Rightarrow \perp$
 $\quad | [Scr] \Rightarrow Scr\ e-id)$

definition $userHasAccess :: user \Rightarrow entry \Rightarrow bool$ **where**
 $userHasAccess\ u\ e = ((fst\ e) = Open \vee (fst\ (snd\ e) = u))$

definition $readEntrySE :: SEPolicy$ **where**
 $readEntrySE\ x = (case\ x\ of\ (readEntry\ u\ r\ p\ e-id, S) \Rightarrow (case\ getEntry\ S\ p\ e-id\ of$
 $\quad \perp \Rightarrow \perp$
 $\quad | [e] \Rightarrow (if\ (userHasAccess\ u\ e)$
 $\quad \quad then\ [allow\ ()]$
 $\quad \quad else\ [deny\ ()]))$
 $\quad | x \Rightarrow \perp)$

definition $deleteEntrySE :: SEPolicy$ **where**
 $deleteEntrySE\ x = (case\ x\ of\ (deleteEntry\ u\ r\ p\ e-id, S) \Rightarrow (case\ getEntry\ S\ p\ e-id\ of$
 $\quad \perp \Rightarrow \perp$
 $\quad | [e] \Rightarrow (if\ (userHasAccess\ u\ e)$
 $\quad \quad then\ [allow\ ()]$
 $\quad \quad else\ [deny\ ()]))$
 $\quad | x \Rightarrow \perp)$

definition $editEntrySE :: SEPolicy$ **where**
 $editEntrySE\ x = (case\ x\ of\ (editEntry\ u\ r\ p\ e-id\ s, S) \Rightarrow (case\ getEntry\ S\ p\ e-id\ of$
 $\quad \perp \Rightarrow \perp$
 $\quad | [e] \Rightarrow (if\ (userHasAccess\ u\ e)$
 $\quad \quad then\ [allow\ ()]$
 $\quad \quad else\ [deny\ ()]))$

| $x \Rightarrow \perp$)

definition *SEPolicy* :: *SEPolicy* **where**

SEPolicy = *editEntrySE* \oplus *deleteEntrySE* \oplus *readEntrySE* \oplus *A_U*

lemmas *SEsimps* = *SEPolicy-def* *get-entry-def* *userHasAccess-def*
editEntrySE-def *deleteEntrySE-def* *readEntrySE-def*

Legitimate Relationships

type-synonym *LRPolicy* = (*Operation* \times Σ , *unit*) *policy*

fun *hasLR*:: *user* \Rightarrow *patient* \Rightarrow Σ \Rightarrow *bool* **where**

hasLR *u p* Σ = (*case* Σ *p* *of* $\perp \Rightarrow$ *False*
| [*lrs*] \Rightarrow (\exists *lr*. *lr* \in (*ran* *lrs*) \wedge *u* \in *lr*))

definition *LRPolicy* :: *LRPolicy* **where**

LRPolicy = ($\lambda(x,y)$. (*if* *hasLR* (*userOfOp* *x*) (*patientOfOp* *x*) *y*
then [*allow* ()]
else [*deny* ()]))

definition *createSCRPolicy* :: *LRPolicy* **where**

createSCRPolicy *x* = (*if* (*is-createSCR* (*fst* *x*))
then [*allow* ()]
else \perp)

definition *addLRPolicy* :: *LRPolicy* **where**

addLRPolicy *x* = (*if* (*is-addLR* (*fst* *x*))
then [*allow* ()]
else \perp)

definition *LR-Policy* **where** *LR-Policy* = *createSCRPolicy* \oplus *addLRPolicy* \oplus *LR-Policy* \oplus *A_U*

lemmas *LRsimps* = *LR-Policy-def* *createSCRPolicy-def* *addLRPolicy-def* *LRPolicy-def*

type-synonym *FunPolicy* = (*Operation* \times *DB* \times Σ , *unit*) *policy*

fun *createFunPolicy* :: *FunPolicy* **where**

createFunPolicy ((*createSCR* *u r p*), (*D*, *S*)) = (*if* *p* \in *dom* *D*
then [*deny* ()]
else [*allow* ()])

| *createFunPolicy* *x* = \perp

```

fun addLRFunPolicy :: FunPolicy where
    addLRFunPolicy ((addLR u r p l us),(D,S)) = (if l ∈ dom S
                                                then [deny ()]
                                                else [allow ()])
    |addLRFunPolicy x = ⊥

fun removeLRFunPolicy :: FunPolicy where
    removeLRFunPolicy ((removeLR u r p l),(D,S)) = (if l ∈ dom S
                                                then [allow ()]
                                                else [deny ()])
    |removeLRFunPolicy x = ⊥

fun readSCRFunPolicy :: FunPolicy where
    readSCRFunPolicy ((readSCR u r p),(D,S)) = (if p ∈ dom D
                                                then [allow ()]
                                                else [deny ()])
    |readSCRFunPolicy x = ⊥

fun deleteSCRFunPolicy :: FunPolicy where
    deleteSCRFunPolicy ((deleteSCR u r p),(D,S)) = (if p ∈ dom D
                                                then [allow ()]
                                                else [deny ()])
    |deleteSCRFunPolicy x = ⊥

fun changeStatusFunPolicy :: FunPolicy where
    changeStatusFunPolicy (changeStatus u r p e s,(d,S)) =
        (case d p of [x] ⇒ (if e ∈ dom x
                            then [allow ()]
                            else [deny ()])
         | - ⇒ [deny ()])
    |changeStatusFunPolicy x = ⊥

fun deleteEntryFunPolicy :: FunPolicy where
    deleteEntryFunPolicy (deleteEntry u r p e,(d,S)) =
        (case d p of [x] ⇒ (if e ∈ dom x
                            then [allow ()]
                            else [deny ()])
         | - ⇒ [deny ()])
    |deleteEntryFunPolicy x = ⊥

fun readEntryFunPolicy :: FunPolicy where
    readEntryFunPolicy (readEntry u r p e,(d,S)) =
        (case d p of [x] ⇒ (if e ∈ dom x

```

$$\begin{aligned}
& \text{then } \lfloor \text{allow } () \rfloor \\
& \text{else } \lfloor \text{deny } () \rfloor \\
& | - \Rightarrow \lfloor \text{deny } () \rfloor \\
\text{readEntryFunPolicy } x = \perp
\end{aligned}$$

fun *appendEntryFunPolicy* :: *FunPolicy* **where**
appendEntryFunPolicy (*appendEntry u r p e ed*,(*d,S*)) =
 (case *d p* of [*x*] \Rightarrow (if $e \in \text{dom } x$
 then $\lfloor \text{deny } () \rfloor$
 else $\lfloor \text{allow } () \rfloor$)
 | - \Rightarrow $\lfloor \text{deny } () \rfloor$)
appendEntryFunPolicy *x* = \perp

fun *editEntryFunPolicy* :: *FunPolicy* **where**
editEntryFunPolicy (*editEntry u r p ei e*,(*d,S*)) =
 (case *d p* of [*x*] \Rightarrow (if $ei \in \text{dom } x$
 then $\lfloor \text{allow } () \rfloor$
 else $\lfloor \text{deny } () \rfloor$)
 | - \Rightarrow $\lfloor \text{deny } () \rfloor$)
editEntryFunPolicy *x* = \perp

definition *FunPolicy* **where**

$$\begin{aligned}
\text{FunPolicy} = & \text{editEntryFunPolicy} \oplus \text{appendEntryFunPolicy} \oplus \\
& \text{readEntryFunPolicy} \oplus \text{deleteEntryFunPolicy} \oplus \\
& \text{changeStatusFunPolicy} \oplus \text{deleteSCRFunPolicy} \oplus \\
& \text{removeLRFunPolicy} \oplus \text{readSCRFunPolicy} \oplus \\
& \text{addLRFunPolicy} \oplus \text{createFunPolicy} \oplus A_U
\end{aligned}$$

Modelling Core RBAC

type-synonym *RBACPolicy* = *Operation* \times *v* \mapsto *unit*

definition *RBAC* :: (*role* \times *Operation*) *set* **where**

$$\begin{aligned}
\text{RBAC} = & \{(r,f). r = \text{Nurse} \wedge \text{is-readEntry } f\} \cup \\
& \{(r,f). r = \text{Nurse} \wedge \text{is-readSCR } f\} \cup \\
& \{(r,f). r = \text{ClinicalPractitioner} \wedge \text{is-appendEntry } f\} \cup \\
& \{(r,f). r = \text{ClinicalPractitioner} \wedge \text{is-deleteEntry } f\} \cup \\
& \{(r,f). r = \text{ClinicalPractitioner} \wedge \text{is-readEntry } f\} \cup \\
& \{(r,f). r = \text{ClinicalPractitioner} \wedge \text{is-readSCR } f\} \cup \\
& \{(r,f). r = \text{ClinicalPractitioner} \wedge \text{is-changeStatus } f\} \cup \\
& \{(r,f). r = \text{ClinicalPractitioner} \wedge \text{is-editEntry } f\} \cup \\
& \{(r,f). r = \text{Clerical} \wedge \text{is-createSCR } f\} \cup \\
& \{(r,f). r = \text{Clerical} \wedge \text{is-deleteSCR } f\} \cup \\
& \{(r,f). r = \text{Clerical} \wedge \text{is-addLR } f\} \cup
\end{aligned}$$

$$\{(r,f). r = \text{Clerical} \wedge \text{is-removeLR } f\}$$

definition *RBACPolicy* :: *RBACPolicy* **where**

RBACPolicy = $(\lambda (f,uc).$

 if $((\text{roleOfOp } f,f) \in \text{RBAC} \wedge [\text{roleOfOp } f] = uc (\text{userOfOp } f))$

 then $[\text{allow } ()]$

 else $[\text{deny } ()]$)

3.1.4 The State Transitions and Output Function

State Transition

fun *OpSuccessDB* :: $(\text{Operation} \times \text{DB}) \rightarrow \text{DB}$ **where**

OpSuccessDB (*createSCR* *u r p*,*S*) = $(\text{case } S \text{ p of } \perp \Rightarrow [S(p \mapsto \emptyset)]$
 $| [x] \Rightarrow [S])$

$| \text{OpSuccessDB } ((\text{appendEntry } u \text{ r p } ei \text{ e}),S) =$
 $(\text{case } S \text{ p of } \perp \Rightarrow [S]$
 $| [x] \Rightarrow ((\text{if } ei \in (\text{dom } x)$
 $\text{then } [S]$
 $\text{else } [S(p \mapsto x(ei \mapsto e))]))))$

$| \text{OpSuccessDB } ((\text{deleteSCR } u \text{ r p}),S) = (\text{Some } (S(p := \perp)))$

$| \text{OpSuccessDB } ((\text{deleteEntry } u \text{ r p } ei),S) =$
 $(\text{case } S \text{ p of } \perp \Rightarrow [S]$
 $| [x] \Rightarrow \text{Some } (S(p \mapsto (x(ei := \perp))))))$

$| \text{OpSuccessDB } ((\text{changeStatus } u \text{ r p } ei \text{ s}),S) =$
 $(\text{case } S \text{ p of } \perp \Rightarrow [S]$
 $| [x] \Rightarrow (\text{case } x \text{ ei of}$
 $\quad [e] \Rightarrow [S(p \mapsto x(ei \mapsto (s, \text{snd } e)))]$
 $\quad | \perp \Rightarrow [S]))$

$| \text{OpSuccessDB } ((\text{editEntry } u \text{ r p } ei \text{ e}),S) =$
 $(\text{case } S \text{ p of } \perp \Rightarrow [S]$
 $| [x] \Rightarrow (\text{case } x \text{ ei of}$
 $\quad [e] \Rightarrow [S(p \mapsto (x(ei \mapsto (e))))]$
 $\quad | \perp \Rightarrow [S]))$

$| \text{OpSuccessDB } (x,S) = [S]$

fun *OpSuccessSigma* :: $(\text{Operation} \times \Sigma) \rightarrow \Sigma$ **where**

OpSuccessSigma (*addLR* *u r p lr-id us*,*S*) =
 $(\text{case } S \text{ p of } [lrs] \Rightarrow (\text{case } (lrs \text{ lr-id}) \text{ of}$
 $\quad \perp \Rightarrow [S(p \mapsto (lrs(lr-id \mapsto us)))]$
 $\quad | [x] \Rightarrow [S])$
 $| \perp \Rightarrow [S(p \mapsto (\text{Map.empty}(lr-id \mapsto us)))]))$

$| \text{OpSuccessSigma } (\text{removeLR } u \text{ r p } lr-id,S) =$

$$\begin{aligned} & (\text{case } S \text{ p of Some lrs} \Rightarrow \lfloor S(p \mapsto (\text{lrs}(lr\text{-id} := \perp))) \rfloor \\ & \quad \mid \perp \Rightarrow \lfloor S \rfloor) \\ \text{OpSuccessSigma } (x, S) &= \lfloor S \rfloor \end{aligned}$$

fun *OpSuccessUC* :: (*Operation* × *v*) → *v* **where**
OpSuccessUC (*f*, *u*) = $\lfloor u \rfloor$

Output

type-synonym *Output* = *unit*

fun *OpSuccessOutput* :: (*Operation*) → *Output* **where**
OpSuccessOutput *x* = $\lfloor () \rfloor$

fun *OpFailOutput* :: *Operation* → *Output* **where**
OpFailOutput *x* = $\lfloor () \rfloor$

3.1.5 Combine All Parts

definition *SE-LR-Policy* :: (*Operation* × *DB* × Σ , *unit*) *policy* **where**
SE-LR-Policy = $(\lambda(x, x). x) \text{ of } (SEPolicy \otimes_{\vee D} LR\text{-Policy}) \text{ o } (\lambda(a, b, c). ((a, b), a, c))$

definition *SE-LR-FUN-Policy* :: (*Operation* × *DB* × Σ , *unit*) *policy* **where**
SE-LR-FUN-Policy = $((\lambda(x, x). x) \text{ of } (FunPolicy \otimes_{\vee D} SE\text{-LR-Policy}) \text{ o } (\lambda a. (a, a)))$

definition *SE-LR-RBAC-Policy* :: (*Operation* × *DB* × Σ × *v*, *unit*) *policy* **where**
SE-LR-RBAC-Policy = $(\lambda(x, x). x) \text{ of } (RBACPolicy \otimes_{\vee D} SE\text{-LR-FUN-Policy}) \text{ o } (\lambda(a, b, c, d). ((a, d), (a, b, c)))$

definition *ST-Allow* :: *Operation* × *DB* × Σ × *v* → *Output* × *DB* × Σ × *v*
where *ST-Allow* = $((OpSuccessOutput \otimes_M (OpSuccessDB \otimes_S OpSuccessSigma \otimes_S OpSuccessUC)) \text{ o } ((\lambda(a, b, c). ((a), (a, b, c))))))$

definition *ST-Deny* :: *Operation* × *DB* × Σ × *v* → *Output* × *DB* × Σ × *v*
where *ST-Deny* = $(\lambda (ope, sp, si, uc). Some ((), sp, si, uc))$

definition *SE-LR-RBAC-ST-Policy* :: *Operation* × *DB* × Σ × *v* → *Output* × *DB* ×

$\Sigma \times v$
where $SE\text{-LR}\text{-RBAC}\text{-ST}\text{-Policy} = ((\lambda (x,y).y)$
 $o_f ((ST\text{-Allow},ST\text{-Deny}) \otimes_{\nabla} SE\text{-LR}\text{-RBAC}\text{-Policy})$
 $o (\lambda x.(x,x)))$

definition $PolMon :: Operation \Rightarrow (Output\ decision, DB \times \Sigma \times v) MON_{SE}$
where $PolMon = (policy2MON\ SE\text{-LR}\text{-RBAC}\text{-ST}\text{-Policy})$

end

3.2 Instantiating Our Secure Service Example

theory

ServiceExample

imports

Service

begin

In the following, we briefly present an instantiations of our secure service example from the last section. We assume three different members of the health care staff and two patients:

3.2.1 Access Control Configuration

definition $alice :: user$ **where** $alice = 1$

definition $bob :: user$ **where** $bob = 2$

definition $charlie :: user$ **where** $charlie = 3$

definition $patient1 :: patient$ **where** $patient1 = 5$

definition $patient2 :: patient$ **where** $patient2 = 6$

definition $UC0 :: v$ **where**

$UC0 = Map.empty(alice \mapsto Nurse, bob \mapsto ClinicalPractitioner, charlie \mapsto Clerical)$

definition $entry1 :: entry$ **where**

$entry1 = (Open,alice, dummyContent)$

definition $entry2 :: entry$ **where**

$entry2 = (Closed,bob, dummyContent)$

definition $entry3 :: entry$ **where**

$entry3 = (Closed,alice, dummyContent)$

definition $SCR1 :: SCR$ **where**

$SCR1 = (Map.empty(1 \mapsto entry1))$

definition *SCR2* :: *SCR* **where**

SCR2 = (*Map.empty*)

definition *Spine0* :: *DB* **where**

Spine0 = *Map.empty*(*patient1* ↦ *SCR1*, *patient2* ↦ *SCR2*)

definition *LR1* :: *LR* **where**

LR1 = (*Map.empty*(1 ↦ {*alice*}))

definition $\Sigma 0$:: Σ **where**

$\Sigma 0$ = (*Map.empty*(*patient1* ↦ *LR1*))

3.2.2 The Initial System State

definition $\sigma 0$:: *DB* × Σ × *v* **where**

$\sigma 0$ = (*Spine0*, $\Sigma 0$, *UC0*)

3.2.3 Basic Properties

lemma [*simp*]: (*case a of allow d* ⇒ [*X*] | *deny d2* ⇒ [*Y*]) = ⊥ ⇒ *False*

by (*case-tac a, simp-all*)

lemma [*cong, simp*]:

((*if hasLR urp1-alice 1* $\Sigma 0$ then [*allow* ()] else [*deny* ()]) = ⊥) = *False*

by (*simp*)

lemmas *MonSimps* = *valid-SE-def unit-SE-def bind-SE-def*

lemmas *Psplits* = *option.splits unit.splits prod.splits decision.splits*

lemmas *PolSimps* = *valid-SE-def unit-SE-def bind-SE-def if-splits policy2MON-def*
SE-LR-RBAC-ST-Policy-def map-add-def id-def LRsimps prod-2-def
RBACPolicy-def

SE-LR-Policy-def SEPolicy-def RBAC-def deleteEntrySE-def editEntrySE-def

readEntrySE-def $\sigma 0$ -*def* $\Sigma 0$ -*def* *UC0-def patient1-def patient2-def LR1-def*

alice-def bob-def charlie-def get-entry-def SE-LR-RBAC-Policy-def Allow-def

Deny-def dom-restrict-def policy-range-comp-def prod-orA-def prod-orD-def

ST-Allow-def ST-Deny-def Spine0-def SCR1-def SCR2-def entry1-def
entry2-def

entry3-def FunPolicy-def SE-LR-FUN-Policy-def o-def image-def UPFDefs

lemma *SE-LR-RBAC-Policy* ((*createSCR* *alice Clerical patient1*), $\sigma\theta$)= *Some* (*deny* ())
by (*simp* *add: PolSimps*)

lemma *exBool*[*simp*]: $\exists a::\text{bool}. a$
by *auto*

lemma *deny-allow*[*simp*]: [*deny* ()] \notin *Some* ‘*range allow*
by *auto*

lemma *allow-deny*[*simp*]: [*allow* ()] \notin *Some* ‘*range deny*
by *auto*

Policy as monad. Alice using her first urp can read the SCR of patient1.

lemma
 $(\sigma\theta \models (os \leftarrow \text{mbind } [(createSCR \text{ alice Clerical patient1})] (PolMon);$
 $(return (os = [(deny (Out)]))))))$
by (*simp* *add: PolMon-def MonSimps PolSimps*)

Presenting her other urp, she is not allowed to read it.

lemma *SE-LR-RBAC-Policy* ((*appendEntry* *alice Clerical patient1 ei d*), $\sigma\theta$)= [*deny* ()]
by (*simp* *add: PolSimps*)

end

4 Conclusion and Related Work

4.1 Related Work

With Barker [3], our UPF shares the observation that a broad range of access control models can be reduced to a surprisingly small number of primitives together with a set of combinators or relations to build more complex policies. We also share the vision that the semantics of access control models should be formally defined. In contrast to [3], UPF uses higher-order constructs and, more importantly, is geared towards machine support for (formally) transforming policies and supporting model-based test case generation approaches.

4.2 Conclusion Future Work

We have presented a uniform framework for modelling security policies. This might be regarded as merely an interesting academic exercise in the art of abstraction, especially given the fact that underlying core concepts are logically equivalent, but presented remarkably different from—apparently simple—security textbook formalisations. However, we have successfully used the framework to model fully the large and complex information governance policy of a national health-care record system as described in the official documents [10] as well as network policies [12]. Thus, we have shown the framework being able to accommodate relatively conventional RBAC [20] mechanisms alongside less common ones such as Legitimate Relationships. These security concepts are modelled separately and combined into one global access control mechanism. Moreover, we have shown the practical relevance of our model by using it in our test generation system HOL-TestGen [9], translating informal security requirements into formal test specifications to be processed to test sequences for a distributed system consisting of applications accessing a central record storage system.

Besides applying our framework to other access control models, we plan to develop specific test case generation algorithms. Such domain-specific algorithms allow, by exploiting knowledge about the structure of access control models, respectively the UPF, for a deeper exploration of the test space. Finally, this results in an improved test coverage.

5 Appendix

5.1 Basic Monad Theory for Sequential Computations

```
theory
  Monads
  imports
    Main
begin
```

5.1.1 General Framework for Monad-based Sequence-Test

As such, Higher-order Logic as a purely functional specification formalism has no built-in mechanism for state and state-transitions. Forms of testing involving state require therefore explicit mechanisms for their treatment inside the logic; a well-known technique to model states inside purely functional languages are *monads* made popular by Wadler and Moggi and extensively used in Haskell. HOL is powerful enough to represent the most important standard monads; however, it is not possible to represent monads as such due to well-known limitations of the Hindley-Milner type-system.

Here is a variant for state-exception monads, that models precisely transition functions with preconditions. Next, we declare the state-backtrack-monad. In all of them, our concept of i/o-stepping functions can be formulated; these are functions mapping input to a given monad. Later on, we will build the usual concepts of:

1. deterministic i/o automata,
2. non-deterministic i/o automata, and
3. labelled transition systems (LTS)

State Exception Monads

type-synonym $(\prime o, \prime \sigma) \text{ MON}_{SE} = \prime \sigma \rightarrow (\prime o \times \prime \sigma)$

definition $\text{bind-SE} :: (\prime o, \prime \sigma) \text{ MON}_{SE} \Rightarrow (\prime o \Rightarrow (\prime o', \prime \sigma) \text{ MON}_{SE}) \Rightarrow (\prime o', \prime \sigma) \text{ MON}_{SE}$

where $\text{bind-SE } f g = (\lambda \sigma. \text{case } f \sigma \text{ of } \text{None} \Rightarrow \text{None}$
 $\quad \quad \quad | \text{Some } (out, \sigma') \Rightarrow g \text{ out } \sigma')$

notation $\text{bind-SE } (\langle \text{bind}_{SE} \rangle)$

syntax

$-bind-SE :: [pttrn, ('o, 'σ)MON_{SE}, ('o', 'σ)MON_{SE}] ⇒ ('o', 'σ)MON_{SE}$
 $(\langle (2 - \leftarrow -; -) \rangle [5, 8, 8] 8)$

syntax-consts

$-bind-SE ⇒ bind-SE$

translations

$x \leftarrow f; g ⇒ CONST bind-SE f (\% x . g)$

definition $unit-SE :: 'o ⇒ ('o, 'σ)MON_{SE} \ (\langle return \ - \rangle 8)$

where $unit-SE \ e = (\lambda \sigma. Some(e, \sigma))$

notation $unit-SE \ (\langle unit_{SE} \rangle)$

definition $fail_{SE} :: ('o, 'σ)MON_{SE}$

where $fail_{SE} = (\lambda \sigma. None)$

notation $fail_{SE} \ (\langle fail_{SE} \rangle)$

definition $assert-SE :: ('σ ⇒ bool) ⇒ (bool, 'σ)MON_{SE}$

where $assert-SE \ P = (\lambda \sigma. if \ P \ \sigma \ then \ Some(True, \sigma) \ else \ None)$

notation $assert-SE \ (\langle assert_{SE} \rangle)$

definition $assume-SE :: ('σ ⇒ bool) ⇒ (unit, 'σ)MON_{SE}$

where $assume-SE \ P = (\lambda \sigma. if \ \exists \sigma . P \ \sigma \ then \ Some((), \ SOME \ \sigma . P \ \sigma) \ else \ None)$

notation $assume-SE \ (\langle assume_{SE} \rangle)$

definition $if-SE :: ['σ ⇒ bool, ('α, 'σ)MON_{SE}, ('α, 'σ)MON_{SE}] ⇒ ('α, 'σ)MON_{SE}$

where $if-SE \ c \ E \ F = (\lambda \sigma. if \ c \ \sigma \ then \ E \ \sigma \ else \ F \ \sigma)$

notation $if-SE \ (\langle if_{SE} \rangle)$

The standard monad theorems about unit and associativity:

lemma $bind-left-unit : (x \leftarrow return \ a; k) = k$

apply ($simp \ add: \ unit-SE-def \ bind-SE-def$)

done

lemma $bind-right-unit: (x \leftarrow m; return \ x) = m$

apply ($simp \ add: \ unit-SE-def \ bind-SE-def$)

apply ($rule \ ext$)

subgoal for σ

apply ($case-tac \ m \ \sigma$)

apply ($simp-all$)

done

done

lemma $bind-assoc: (y \leftarrow (x \leftarrow m; k); h) = (x \leftarrow m; (y \leftarrow k; h))$

apply ($simp \ add: \ unit-SE-def \ bind-SE-def$)

apply ($rule \ ext$)

```

subgoal for  $\sigma$ 
  apply (case-tac  $m$   $\sigma$ , simp-all)
  subgoal for  $a$ 
    apply (case-tac  $a$ , simp-all)
    done
  done
done

```

In order to express test-sequences also on the object-level and to make our theory amenable to formal reasoning over test-sequences, we represent them as lists of input and generalize the bind-operator of the state-exception monad accordingly. The approach is straightforward, but comes with a price: we have to encapsulate all input and output data into one type. Assume that we have a typed interface to a module with the operations op_1, op_2, \dots, op_n with the inputs $\iota_1, \iota_2, \dots, \iota_n$ (outputs are treated analogously). Then we can encode for this interface the general input - type:

$$\text{datatype in} = op_1 :: \iota_1 \mid \dots \mid \iota_n$$

Obviously, we loose some type-safety in this approach; we have to express that in traces only *corresponding* input and output belonging to the same operation will occur; this form of side-conditions have to be expressed inside HOL. From the user perspective, this will not make much difference, since junk-data resulting from too weak typing can be ruled out by adopted front-ends.

In order to express test-sequences also on the object-level and to make our theory amenable to formal reasoning over test-sequences, we represent them as lists of input and generalize the bind-operator of the state-exception monad accordingly. Thus, the notion of test-sequence is mapped to the notion of a *computation*, a semantic notion; at times we will use reifications of computations, i.e. a data-type in order to make computation amenable to case-splitting and meta-theoretic reasoning. To this end, we have to encapsulate all input and output data into one type. Assume that we have a typed interface to a module with the operations op_1, op_2, \dots, op_n with the inputs $\iota_1, \iota_2, \dots, \iota_n$ (outputs are treated analogously). Then we can encode for this interface the general input - type:

$$\text{datatype in} = op_1 :: \iota_1 \mid \dots \mid \iota_n$$

Obviously, we loose some type-safety in this approach; we have to express that in traces only *corresponding* input and output belonging to the same operation will occur; this form of side-conditions have to be expressed inside HOL. From the user perspective, this will not make much difference, since junk-data resulting from too weak typing can be ruled out by adopted front-ends.

Note that the subsequent notion of a test-sequence allows the io stepping function (and the special case of a program under test) to stop execution *within* the sequence; such premature terminations are characterized by an output list which is shorter than the input list. Note that our primary notion of multiple execution ignores failure and reports failure steps only by missing results ...

```

fun   mbind :: 'l list ⇒ ('l ⇒ ('o,'σ) MONSE) ⇒ ('o list,'σ) MONSE
where mbind [] iostep σ = Some([], σ) |
      mbind (a#H) iostep σ =
        (case iostep a σ of
         None   ⇒ Some([], σ)
        | Some (out, σ') ⇒ (case mbind H iostep σ' of
                             None   ⇒ Some([out],σ')
                             | Some(outs,σ'') ⇒ Some(out#outs,σ'')))

```

As mentioned, this definition is fail-safe; in case of an exception, the current state is maintained, no result is reported. An alternative is the fail-strict variant $mbind'$ defined below.

```

lemma mbind-unit [simp]: mbind [] f = (return [])
by(rule ext, simp add: unit-SE-def)

```

```

lemma mbind-nofailure [simp]: mbind S f σ ≠ None
apply (rule-tac x=σ in spec)
apply (induct S)
using mbind.simps(1) apply force
apply(simp add:unit-SE-def)
apply(safe)[1]
subgoal for a S x
  apply (case-tac f a x)
  apply(simp)
  apply(safe)[1]
  subgoal for aa b
    apply (erule-tac x=b in allE)
    apply (erule exE)+
    apply (simp)
  done
done
done

```

The fail-strict version of $mbind'$ looks as follows:

```

fun   mbind' :: 'l list ⇒ ('l ⇒ ('o,'σ) MONSE) ⇒ ('o list,'σ) MONSE
where mbind' [] iostep σ = Some([], σ) |
      mbind' (a#H) iostep σ =
        (case iostep a σ of
         None   ⇒ None
        | Some (out, σ') ⇒ (case mbind H iostep σ' of
                             None   ⇒ None — fail-strict
                             | Some(outs,σ'') ⇒ Some(out#outs,σ'')))

```

$mbind'$ as failure strict operator can be seen as a foldr on bind—if the types would

match ...

definition $try\text{-}SE :: ('o, 'σ) MON_{SE} ⇒ ('o\ option, 'σ) MON_{SE}$

where $try\text{-}SE\ ioprogram = (\lambda\sigma. case\ ioprogram\ \sigma\ of$
 $None \Rightarrow Some(None, \sigma)$
 $| Some(outs, \sigma') \Rightarrow Some(Some\ outs, \sigma'))$

In contrast $mbind$ as a failure safe operator can roughly be seen as a $foldr$ on $bind$ - try : $m1 ; try\ m2 ; try\ m3 ; \dots$. Note, that the rough equivalence only holds for certain predicates in the sequence - length equivalence modulo $None$, for example. However, if a conditional is added, the equivalence can be made precise:

lemma $mbind\text{-}try$:

$(x \leftarrow mbind\ (a\#S)\ F; M\ x) =$
 $(a' \leftarrow try\text{-}SE(F\ a);$
 $if\ a' = None$
 $then\ (M\ [])$
 $else\ (x \leftarrow mbind\ S\ F; M\ (the\ a' \# x)))$

apply ($rule\ ext$)

apply ($simp\ add: bind\text{-}SE\text{-}def\ try\text{-}SE\text{-}def$)

subgoal for x

apply ($case\text{-}tac\ F\ a\ x$)

apply ($simp$)

apply ($safe$)[1]

apply ($simp\ add: bind\text{-}SE\text{-}def\ try\text{-}SE\text{-}def$)

subgoal for $aa\ b$

apply ($case\text{-}tac\ mbind\ S\ F\ b$)

apply ($auto$)

done

done

done

On this basis, a symbolic evaluation scheme can be established that reduces $mbind$ -code to $try\text{-}SE$ -code and If-cascades.

definition $alt\text{-}SE :: [('o, 'σ)MON_{SE}, ('o, 'σ)MON_{SE}] ⇒ ('o, 'σ)MON_{SE}$ (**infixl** $\langle \sqcap_{SE} \rangle\ 10$)

where $(f\ \sqcap_{SE}\ g) = (\lambda\sigma. case\ f\ \sigma\ of\ None \Rightarrow g\ \sigma$
 $| Some\ H \Rightarrow Some\ H)$

definition $malt\text{-}SE :: ('o, 'σ)MON_{SE}\ list \Rightarrow ('o, 'σ)MON_{SE}$

where $malt\text{-}SE\ S = foldr\ alt\text{-}SE\ S\ fail_{SE}$

notation $malt\text{-}SE\ \langle \sqcap_{SE} \rangle$

lemma $malt\text{-}SE\text{-}mt\ [simp]: \sqcap_{SE}\ [] = fail_{SE}$

by ($simp\ add: malt\text{-}SE\text{-}def$)

lemma *malt-SE-cons* [*simp*]: $\prod_{SE} (a \# S) = (a \sqcap_{SE} (\prod_{SE} S))$
by (*simp add: malt-SE-def*)

State-Backtrack Monads

This subsection is still rudimentary and as such an interesting formal analogue to the previous monad definitions. It is doubtful that it is interesting for testing and as a computational structure at all. Clearly more relevant is “sequence” instead of “set,” which would rephrase Isabelle’s internal tactic concept.

type-synonym $(\prime o, \prime \sigma) MON_{SB} = \prime \sigma \Rightarrow (\prime o \times \prime \sigma) \text{ set}$

definition *bind-SB* :: $(\prime o, \prime \sigma) MON_{SB} \Rightarrow (\prime o \Rightarrow (\prime o', \prime \sigma) MON_{SB}) \Rightarrow (\prime o', \prime \sigma) MON_{SB}$
where $bind-SB f g \sigma = \bigcup ((\lambda(out, \sigma). (g \text{ out } \sigma)) \text{ ` } (f \sigma))$

notation *bind-SB* ($\langle bind_{SB} \rangle$)

definition *unit-SB* :: $\prime o \Rightarrow (\prime o, \prime \sigma) MON_{SB} (\langle returns \text{ - } \rangle \delta)$

where $unit-SB e = (\lambda \sigma. \{(e, \sigma)\})$

notation *unit-SB* ($\langle unit_{SB} \rangle$)

syntax *-bind-SB* :: $[pttrn, (\prime o, \prime \sigma) MON_{SB}, (\prime o', \prime \sigma) MON_{SB}] \Rightarrow (\prime o', \prime \sigma) MON_{SB}$
 $(\langle (2 \text{ - } := \text{ - } \text{ - } \rangle [5, 8, 8] \delta)$

syntax-consts *-bind-SB* $\equiv bind-SB$

translations

$x := f; g \equiv CONST \text{ bind-SB } f (\% x . g)$

lemma *bind-left-unit-SB* : $(x := returns a; m) = m$

apply (*rule ext*)

apply (*simp add: unit-SB-def bind-SB-def*)

done

lemma *bind-right-unit-SB*: $(x := m; returns x) = m$

apply (*rule ext*)

apply (*simp add: unit-SB-def bind-SB-def*)

done

lemma *bind-assoc-SB*: $(y := (x := m; k); h) = (x := m; (y := k; h))$

apply (*rule ext*)

apply (*simp add: unit-SB-def bind-SB-def split-def*)

done

State Backtrack Exception Monad

The following combination of the previous two Monad-Constructions allows for the semantic foundation of a simple generic assertion language in the style of Schirmer’s Simpl-

Language or Rustan Leino's Boogie-PL language. The key is to use the exceptional element `None` for violations of the assert-statement.

type-synonym $(\text{'}o, \text{'}\sigma) \text{MON}_{SBE} = \text{'}\sigma \Rightarrow ((\text{'}o \times \text{'}\sigma) \text{set}) \text{option}$

definition $\text{bind-SBE} :: (\text{'}o, \text{'}\sigma) \text{MON}_{SBE} \Rightarrow (\text{'}o \Rightarrow (\text{'}o', \text{'}\sigma) \text{MON}_{SBE}) \Rightarrow (\text{'}o', \text{'}\sigma) \text{MON}_{SBE}$

where $\text{bind-SBE } f g = (\lambda\sigma. \text{case } f \text{ } \sigma \text{ of } \text{None} \Rightarrow \text{None}$
 $\quad \quad \quad | \text{Some } S \Rightarrow (\text{let } S' = (\lambda(\text{out}, \sigma'). g \text{ out } \sigma') \text{ ' } S$
 $\quad \quad \quad \text{in if } \text{None} \in S' \text{ then } \text{None}$
 $\quad \quad \quad \text{else } \text{Some}(\bigcup (\text{the ' } S'))))$

syntax $\text{-bind-SBE} :: [\text{pttrn}, (\text{'}o, \text{'}\sigma) \text{MON}_{SBE}, (\text{'}o', \text{'}\sigma) \text{MON}_{SBE}] \Rightarrow (\text{'}o', \text{'}\sigma) \text{MON}_{SBE}$
 $\quad \quad \quad (\langle (2 - := -; -) \rangle [5, 8, 8] 8)$

syntax-consts $\text{-bind-SBE} \Rightarrow \text{bind-SBE}$

translations

$x := f; g \Rightarrow \text{CONST } \text{bind-SBE } f (\% x . g)$

definition $\text{unit-SBE} :: \text{'}o \Rightarrow (\text{'}o, \text{'}\sigma) \text{MON}_{SBE} \quad (\langle (\text{returning } -) \rangle 8)$

where $\text{unit-SBE } e = (\lambda\sigma. \text{Some}(\{(e, \sigma)\}))$

definition $\text{assert-SBE} :: (\text{'}\sigma \Rightarrow \text{bool}) \Rightarrow (\text{unit}, \text{'}\sigma) \text{MON}_{SBE}$

where $\text{assert-SBE } e = (\lambda\sigma. \text{if } e \text{ } \sigma \text{ then } \text{Some}(\{((\text{ }, \sigma))\})$
 $\quad \quad \quad \text{else } \text{None})$

notation $\text{assert-SBE} \quad (\langle \text{assert}_{SBE} \rangle)$

definition $\text{assume-SBE} :: (\text{'}\sigma \Rightarrow \text{bool}) \Rightarrow (\text{unit}, \text{'}\sigma) \text{MON}_{SBE}$

where $\text{assume-SBE } e = (\lambda\sigma. \text{if } e \text{ } \sigma \text{ then } \text{Some}(\{((\text{ }, \sigma))\})$
 $\quad \quad \quad \text{else } \text{Some} \{ \})$

notation $\text{assume-SBE} \quad (\langle \text{assume}_{SBE} \rangle)$

definition $\text{havoc-SBE} :: (\text{unit}, \text{'}\sigma) \text{MON}_{SBE}$

where $\text{havoc-SBE} = (\lambda\sigma. \text{Some}(\{x. \text{True}\}))$

notation $\text{havoc-SBE} \quad (\langle \text{havoc}_{SBE} \rangle)$

lemma $\text{bind-left-unit-SBE} : (x := \text{returning } a; m) = m$

apply (rule ext)

apply $(\text{simp add: unit-SBE-def bind-SBE-def})$

done

lemma $\text{bind-right-unit-SBE} : (x := m; \text{returning } x) = m$

apply (rule ext)

apply $(\text{simp add: unit-SBE-def bind-SBE-def})$

subgoal for x

```

apply (case-tac m x)
  apply (simp-all add:Let-def)
apply (rule HOL.ccontr)
apply (simp add: Set.image-iff)
done
done

lemmas aux = trans[OF HOL.neq-commute, OF Option.not-None-eq]

lemma bind-assoc-SBE: (y := (x := m; k); h) = (x := m; (y := k; h))
proof (rule ext, simp add: unit-SBE-def bind-SBE-def, rename-tac x,
  case-tac m x, simp-all add: Let-def Set.image-iff, safe, goal-cases)
  case (1 x a aa b ab ba a b)
  then show ?case by(rule-tac x=(a, b) in bexI, simp-all)
next
  case (2 x a aa b ab ba)
  then show ?case
    apply (rule-tac x=(aa, b) in bexI, simp-all add:split-def)
    apply (erule-tac x=(aa,b) in ballE)
    apply (auto simp: aux image-def split-def intro!: rev-bexI)
    done
next
  case (3 x a a b)
  then show ?case by(rule-tac x=(a, b) in bexI, simp-all)
next
  case (4 x a aa b)
  then show ?case
    apply (erule-tac Q=None = X for X in contrapos-pp)
    apply (erule-tac x=(aa,b) and P= $\lambda$  x. None  $\neq$  case-prod ( $\lambda$ out. k) x in ballE)
    apply (auto simp: aux image-def split-def intro!: rev-bexI)
    done
next
  case (5 x a aa b ab ba a b)
  then show ?case apply simp apply ((erule-tac x=(ab,ba) in ballE)+)
    apply (simp-all add: aux, (erule exE)+, simp add:split-def)
    apply (erule rev-bexI, case-tac None $\in$ ( $\lambda$ p. h(snd p))'y, auto simp:split-def)
    done

next
  case (6 x a aa b a b)
  then show ?case apply simp apply ((erule-tac x=(a,b) in ballE)+)
    apply (simp-all add: aux, (erule exE)+, simp add:split-def)
    apply (erule rev-bexI, case-tac None $\in$ ( $\lambda$ p. h(snd p))'y, auto simp:split-def)
    done

```

qed

5.1.2 Valid Test Sequences in the State Exception Monad

This is still an unstructured merge of executable monad concepts and specification oriented high-level properties initiating test procedures.

definition *valid-SE* :: ' $\sigma \Rightarrow (bool, 'a) MON_{SE} \Rightarrow bool$ (**infix** $\langle \models \rangle$ 15)
where $(\sigma \models m) = (m \sigma \neq None \wedge fst(the (m \sigma)))$

This notation considers failures as valid—a definition inspired by I/O conformance. Note that it is not possible to define this concept once and for all in a Hindley-Milner type-system. For the moment, we present it only for the state-exception monad, although for the same definition, this notion is applicable to other monads as well.

lemma *syntax-test* :

$\sigma \models (os \leftarrow (mbind \iota s ioprogram); return(length \iota s = length os))$

oops

lemma *valid-true[simp]*: $(\sigma \models (s \leftarrow return x ; return (P s))) = P x$
by(*simp add: valid-SE-def unit-SE-def bind-SE-def*)

Recall `mbind_unit` for the base case.

lemma *valid-failure*: $ioprogram a \sigma = None \implies$

$(\sigma \models (s \leftarrow mbind (a\#S) ioprogram ; M s)) =$
 $(\sigma \models (M []))$

by(*simp add: valid-SE-def unit-SE-def bind-SE-def*)

lemma *valid-failure'*: $A \sigma = None \implies \neg(\sigma \models ((s \leftarrow A ; M s)))$

by(*simp add: valid-SE-def unit-SE-def bind-SE-def*)

lemma *valid-successElem*:

$M \sigma = Some(f \sigma, \sigma) \implies (\sigma \models M) = f \sigma$

by(*simp add: valid-SE-def unit-SE-def bind-SE-def*)

lemma *valid-success*: $ioprogram a \sigma = Some(b, \sigma') \implies$

$(\sigma \models (s \leftarrow mbind (a\#S) ioprogram ; M s)) =$

$(\sigma' \models (s \leftarrow mbind S ioprogram ; M (b\#s)))$

apply (*simp add: valid-SE-def unit-SE-def bind-SE-def*)

apply (*cases mbind S ioprogram \sigma'*, *auto*)

done

lemma *valid-success''*: $ioprogram a \sigma = Some(b, \sigma') \implies$

```

      ( $\sigma \models (s \leftarrow \text{mbind } (a\#S) \text{ ioprogram ; return } (P s))$ ) =
      ( $\sigma' \models (s \leftarrow \text{mbind } S \text{ ioprogram ; return } (P (b\#s)))$ )
apply (simp add: valid-SE-def unit-SE-def bind-SE-def)
apply (cases mbind S ioprogram  $\sigma'$ )
apply (simp-all)
apply (auto)
done

```

lemma *valid-success'*: $A \sigma = \text{Some}(b, \sigma') \implies (\sigma \models ((s \leftarrow A ; M s))) = (\sigma' \models (M b))$
by(*simp add: valid-SE-def unit-SE-def bind-SE-def*)

```

lemma valid-both: ( $\sigma \models (s \leftarrow \text{mbind } (a\#S) \text{ ioprogram ; return } (P s))$ ) =
      (case ioprogram a  $\sigma$  of
        None  $\implies (\sigma \models (\text{return } (P [])))$ 
        | Some(b,  $\sigma'$ )  $\implies (\sigma' \models (s \leftarrow \text{mbind } S \text{ ioprogram ; return } (P (b\#s))))$ )
apply (case-tac ioprogram a  $\sigma$ )
apply (simp-all add: valid-failure valid-success'' split: prod.splits)
done

```

lemma *valid-propagate-1* [*simp*]: $(\sigma \models (\text{return } P)) = (P)$
by(*auto simp: valid-SE-def unit-SE-def*)

```

lemma valid-propagate-2:  $\sigma \models ((s \leftarrow A ; M s)) \implies \exists v \sigma'. \text{the}(A \sigma) = (v, \sigma') \wedge \sigma' \models$ 
( $M v$ )
apply (auto simp: valid-SE-def unit-SE-def bind-SE-def)
apply (cases A  $\sigma$ )
apply (simp-all)
apply (drule-tac x=A  $\sigma$  and f=the in arg-cong)
apply (simp)
apply (rename-tac a b aa)
apply (rule-tac x=fst aa in exI)
apply (rule-tac x=snd aa in exI)
by (auto)

```

```

lemma valid-propagate-2':  $\sigma \models ((s \leftarrow A ; M s)) \implies \exists a. (A \sigma) = \text{Some } a \wedge (\text{snd } a)$ 
 $\models (M (\text{fst } a))$ 
apply (auto simp: valid-SE-def unit-SE-def bind-SE-def)
apply (cases A  $\sigma$ )
apply (simp-all)
apply (simp-all split: prod.splits)
apply (drule-tac x=A  $\sigma$  and f=the in arg-cong)
apply (simp)
apply (rename-tac a b aa x1 x2)
apply (rule-tac x=fst aa in exI)

```

```

apply (rule-tac x=snd aa in exI)
apply (auto)
done

```

```

lemma valid-propagate-2'':  $\sigma \models ((s \leftarrow A ; M s)) \implies \exists v \sigma'. A \sigma = \text{Some}(v, \sigma') \wedge \sigma' \models (M v)$ 
apply (auto simp: valid-SE-def unit-SE-def bind-SE-def)
apply (cases A  $\sigma$ )
apply (simp-all)
apply (drule-tac x=A  $\sigma$  and f=the in arg-cong)
apply (simp)
apply (rename-tac a b aa )
apply (rule-tac x=fst aa in exI)
apply (rule-tac x=snd aa in exI)
apply (auto)
done

```

```

lemma valid-propoagate-3[simp]:  $(\sigma_0 \models (\lambda\sigma. \text{Some} (f \sigma, \sigma))) = (f \sigma_0)$ 
by(simp add: valid-SE-def )

```

```

lemma valid-propoagate-3'[simp]:  $\neg(\sigma_0 \models (\lambda\sigma. \text{None}))$ 
by(simp add: valid-SE-def )

```

```

lemma assert-disch1 :  $P \sigma \implies (\sigma \models (x \leftarrow \text{assert}_{SE} P ; M x)) = (\sigma \models (M \text{True}))$ 
by(auto simp: bind-SE-def assert-SE-def valid-SE-def)

```

```

lemma assert-disch2 :  $\neg P \sigma \implies \neg (\sigma \models (x \leftarrow \text{assert}_{SE} P ; M s))$ 
by(auto simp: bind-SE-def assert-SE-def valid-SE-def)

```

```

lemma assert-disch3 :  $\neg P \sigma \implies \neg (\sigma \models (\text{assert}_{SE} P))$ 
by(auto simp: bind-SE-def assert-SE-def valid-SE-def)

```

```

lemma assert-D :  $(\sigma \models (x \leftarrow \text{assert}_{SE} P ; M x)) \implies P \sigma \wedge (\sigma \models (M \text{True}))$ 
by(auto simp: bind-SE-def assert-SE-def valid-SE-def split: HOL.if-split-asm)

```

```

lemma assume-D :  $(\sigma \models (x \leftarrow \text{assume}_{SE} P ; M x)) \implies \exists \sigma. (P \sigma \wedge \sigma \models (M ()))$ 
apply (auto simp: bind-SE-def assume-SE-def valid-SE-def split: HOL.if-split-asm)
apply (rule-tac x=Eps P in exI)
apply (auto)[1]
subgoal for x a b
apply (rule-tac x=True in exI, rule-tac x=b in exI)
apply (subst Hilbert-Choice.someI)
apply (assumption)
apply (simp)

```

```

done
apply (subst Hilbert-Choice.someI,assumption)
apply (simp)
done

```

These two rule prove that the SE Monad in connection with the notion of valid sequence is actually sufficient for a representation of a Boogie-like language. The SBE monad with explicit sets of states—to be shown below—is strictly speaking not necessary (and will therefore be discontinued in the development).

```

lemma if-SE-D1 :  $P \sigma \implies (\sigma \models_{SE} P B_1 B_2) = (\sigma \models B_1)$ 
by(auto simp: if-SE-def valid-SE-def)

```

```

lemma if-SE-D2 :  $\neg P \sigma \implies (\sigma \models_{SE} P B_1 B_2) = (\sigma \models B_2)$ 
by(auto simp: if-SE-def valid-SE-def)

```

```

lemma if-SE-split-asm :  $(\sigma \models_{SE} P B_1 B_2) = ((P \sigma \wedge (\sigma \models B_1)) \vee (\neg P \sigma \wedge (\sigma \models B_2)))$ 
by(cases P \sigma, auto simp: if-SE-D1 if-SE-D2)

```

```

lemma if-SE-split :  $(\sigma \models_{SE} P B_1 B_2) = ((P \sigma \longrightarrow (\sigma \models B_1)) \wedge (\neg P \sigma \longrightarrow (\sigma \models B_2)))$ 
by(cases P \sigma, auto simp: if-SE-D1 if-SE-D2)

```

```

lemma [code]:  $(\sigma \models m) = (case (m \sigma) of None \Rightarrow False \mid (Some (x,y)) \Rightarrow x)$ 
apply (simp add: valid-SE-def)
apply (cases m \sigma = None)
apply (simp-all)
apply (insert not-None-eq)
apply (auto)
done

```

5.1.3 Valid Test Sequences in the State Exception Backtrack Monad

This is still an unstructured merge of executable monad concepts and specification oriented high-level properties initiating test procedures.

```

definition valid-SBE ::  $'\sigma \Rightarrow ('a, '\sigma) MON_{SBE} \Rightarrow bool$  (infix  $\langle \models_{SBE} \rangle$  15)
where  $\sigma \models_{SBE} m \equiv (m \sigma \neq None)$ 

```

This notation considers all non-failures as valid.

```

lemma assume-assert:  $(\sigma \models_{SBE} ( - \equiv assume_{SBE} P ; assert_{SBE} Q)) = (P \sigma \longrightarrow Q \sigma)$ 
by(simp add: valid-SBE-def assume-SBE-def assert-SBE-def bind-SBE-def)

```

```

lemma assert-intro:  $Q \sigma \implies \sigma \models_{SBE} (assert_{SBE} Q)$ 

```

by(*simp add: valid-SBE-def assume-SBE-def assert-SBE-def bind-SBE-def*)

end

Bibliography

- [1] *American National Standard for Information Technology – Role Based Access Control*. ANSI, New York, Feb. 2004. ANSI INCITS 359-2004.
- [2] C. A. Ardagna, S. D. C. di Vimercati, S. Foresti, T. W. Grandison, S. Jajodia, and P. Samarati. Access control for smarter healthcare using policy spaces. *Computers & Security*, 2010. ISSN 0167-4048. doi: [10.1016/j.cose.2010.07.001](https://doi.org/10.1016/j.cose.2010.07.001).
- [3] S. Barker. The next 700 access control models or a unifying meta-model? In *Proceedings of the 14th ACM symposium on Access control models and technologies, SACMAT '09*, pages 187–196, New York, NY USA, 2009. ACM Press. ISBN 978-1-60558-537-6. doi: [10.1145/1542207.1542238](https://doi.org/10.1145/1542207.1542238).
- [4] M. Y. Becker. Information governance in nhs’s npfit: A case for policy specification. *International Journal of Medical Informatics*, 76(5-6):432–437, 2007. ISSN 1386-5056. doi: [10.1016/j.ijmedinf.2006.09.008](https://doi.org/10.1016/j.ijmedinf.2006.09.008).
- [5] D. E. Bell. Looking back at the bell-la padula model. pages 337–351, Los Alamitos, CA, USA, 2005. pub-ieee. ISBN 1063-9527. doi: [10.1109/CSAC.2005.37](https://doi.org/10.1109/CSAC.2005.37).
- [6] D. E. Bell and L. J. LaPadula. Secure computer systems: A mathematical model, volume II. In *Journal of Computer Security 4*, pages 229–263, 1996. An electronic reconstruction of *Secure Computer Systems: Mathematical Foundations*, 1973.
- [7] E. Bertino, P. A. Bonatti, and E. Ferrari. Trbac: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001. ISSN 1094-9224. doi: [10.1145/501978.501979](https://doi.org/10.1145/501978.501979).
- [8] A. D. Brucker and H. Petritsch. Extending access control models with break-glass. In B. Carminati and J. Joshi, editors, *ACM symposium on access control models and technologies (SACMAT)*, pages 197–206. ACM Press, New York, NY, USA, 2009. ISBN 978-1-60558-537-6. doi: [10.1145/1542207.1542239](https://doi.org/10.1145/1542207.1542239). URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-extending-2009>.
- [9] A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 25(5):683–721, 2013. ISSN 0934-5043. doi: [10.1007/s00165-012-0222-y](https://doi.org/10.1007/s00165-012-0222-y). URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-theorem-prover-2012>.
- [10] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. An approach to modular and testable security models of real-world health-care applications. In

- ACM symposium on access control models and technologies (SACMAT)*, pages 133–142, New York, NY, USA, 2011. ACM Press. ISBN 978-1-4503-0688-1. doi: 10.1145/1998441.1998461. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-model-based-2011>.
- [11] A. D. Brucker, L. Brügger, and B. Wolff. HOL-TestGen/FW: an environment for specification-based firewall conformance testing. In Z. Liu, J. Woodcock, and H. Zhu, editors, *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, number 8049 in Lecture Notes in Computer Science, pages 112–121. Springer-Verlag, Heidelberg, 2013. ISBN 978-3-642-39717-2. doi: 10.1007/978-3-642-39718-9_7. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-testgen-fw-2013>.
- [12] A. D. Brucker, L. Brügger, and B. Wolff. Formal firewall conformance testing: An application of test and proof techniques. *Software Testing, Verification & Reliability (STVR)*, 2014. doi: 10.1002/stvr.1544. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-formal-fw-testing-2014>.
- [13] L. Brügger. *A Framework for Modelling and Testing of Security Policies*. PhD thesis, ETH Zurich, 2012. URL <http://www.brucker.ch/bibliography/abstract/bruegger-generation-2012>. ETH Dissertation No. 20513.
- [14] A. Ferreira, D. Chadwick, P. Farinha, G. Zhao, R. Chilro, R. Cruz-Correia, and L. Antunes. How to securely break into rbac: the btg-rbac model. In *Annual Computer Security Applications Conference (ACSAC)*, 2009.
- [15] N. Li, J. Byun, and E. Bertino. A critique of the ansi standard on role-based access control. *Security Privacy, IEEE*, 5(6):41–49, nov.-dec. 2007. ISSN 1540-7993. doi: 10.1109/MSP.2007.158.
- [16] M. Moyer and M. Abamad. Generalized role-based access control. *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 391–398, Apr 2001. doi: 10.1109/ICDSC.2001.918969.
- [17] OASIS. eXtensible Access Control Markup Language (XACML), version 2.0, 2005. URL <http://docs.oasis-open.org/xacml/2.0/XACML-2.0-OS-NORMATIVE.zip>.
- [18] A. Samuel, A. Ghafoor, and E. Bertino. Context-aware adaptation of access-control policies. *Internet Computing, IEEE*, 12(1):51–54, 2008. ISSN 1089-7801. doi: 10.1109/MIC.2008.6.
- [19] R. Sandhu, V. Bhamidipati, and Q. Munawer. The arbac97 model for role-based administration of roles. *ACM Transactions on Information and System Security*, 2(1):105–135, 1999. ISSN 1094-9224. doi: 10.1145/300830.300839.
- [20] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996. ISSN 0018-9162. URL [http://ite.gmu.edu/list/journals/computer/pdf_ver/i94rbac\(org\).pdf](http://ite.gmu.edu/list/journals/computer/pdf_ver/i94rbac(org).pdf).

- [21] R. S. Sandhu, D. F. Ferraiolo, and D. R. Kuhn. The nist model for role-based access control: towards a unified standard. In *ACM Workshop on Role-Based Access Control*, pages 47–63, 2000. doi: [10.1145/344287.344301](https://doi.org/10.1145/344287.344301).
- [22] J. Wainer, A. Kumar, and P. Barthelmess. Dw-rbac: A formal security model of delegation and revocation in workflow systems. *Inf. Syst.*, 32(3):365–384, 2007. ISSN 0306-4379. doi: <https://doi.org/10.1016/j.is.2005.11.008>.