The Unified Policy Framework
(UPF)

Achim D. Brucker* Lukas Brügger† Burkhart 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
burkhart.wolff@lri.fr

August 16, 2018

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 14
   2.2.2 Common Instances 16
   2.2.3 Domain, Range, and Restrictions 17
   2.3 Sequential Composition 21
   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 29
   2.4.3 Range Splitting 30
   2.4.4 Distributivity of the parallel combinators 30
   2.5 Properties on Policies 32
   2.5.1 Basic Properties 33
   2.5.2 Combined Data-Policy Refinement 34
   2.5.3 Equivalence of Policies 35
   2.6 Policy Transformations 36
   2.6.1 Elementary Operators 37
   2.6.2 Distributivity of the Transformation 40
   2.7 Policy Transformation for Testing 45
   2.8 Putting Everything Together: UPF 47

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
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 $\alpha$ (arguments, system state, security context, ...) to output data $\beta$:

\begin{verbatim}
datatype 'a decision = allow 'a | deny 'a

type-synonym ('a,'b) policy = 'a -> 'b decision (infixr |-> 0)
\end{verbatim}

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

\begin{verbatim}
translations (type)  'a |-> 'b <= (type) 'a -> 'b decision

type-notation policy (infixr |<-> 0)
\end{verbatim}
... 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 (⊥)
- **notation** Some \([\cdot] 80\)  

Thus, the range of a policy may consist of \( \text{accept } x \) data, of \( \text{deny } x \) data, as well as \( \bot \) 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 \( \text{Allow} \) and the deny-set \( \text{Deny} \) (written \( A \) and \( D \) set for short), to characterize these two main sets of the range of a policy.

- **definition** \( \text{Allow} :: (\alpha \text{ decision}) \text{ set} \)
  - **where** \( \text{Allow} = \text{range allow} \)
- **definition** \( \text{Deny} :: (\alpha \text{ decision}) \text{ set} \)
  - **where** \( \text{Deny} = \text{range deny} \)

### 2.1.2 Policy Constructors

Most elementary policy constructors are based on the update operation \( \text{Fun.fun-upd-def} \) \( \text{Fun.upd-def} \):

\[
?f(x := ?a) = (\lambda x. \text{if } x = ?a \text{ then } ?b \text{ else } ?f x)
\]

and the maplet-notation \( a(x \mapsto y) \).

Furthermore, we add notation adopted to our problem domain:

**nonterminal** \( \text{policylets and policylet} \)

**syntax**

- \( \text{policylet1} :: [a, a] => \text{policylet} \)
- \( \text{policylet2} :: [a, a] => \text{policylet} \)
- \( \text{Maplets} :: [\text{policylet}, \text{policylets}] => \text{policylets} \)
- \( \text{Maplets} :: [\text{policylet}, \text{policylets}] => \text{policylets} \)
- \( \text{MapUpd} :: [a => b, \text{policylets}] => a => b \)
- \( \text{emptypolicy} :: a => b \)  

**translations**

- \( \text{MapUpd } m \text{ (Maplets } xy \text{ ms) } \Rightarrow \text{MapUpd } (\text{MapUpd } m \text{ xy) ms} \)
- \( \text{MapUpd } m \text{ (policylet1 } x \text{ y) } \Rightarrow m(x := \text{CONST Some (CONST allow y))} \)
- \( \text{MapUpd } m \text{ (policylet2 } x \text{ y) } \Rightarrow m(x := \text{CONST Some (CONST deny y))} \)
- \( \emptyset \Rightarrow \text{CONST Map.empty} \)

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 \text{test2}: \( p(x \mapsto_+ a, x \mapsto_- b) = p(x \mapsto_- b) \) \text{ by simp}

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

lemma \text{pol-upd-triv1}: t k = \lfloor \text{allow } x \rfloor \implies t(k \mapsto_+ x) = t
\text{ by (rule ext) simp}

lemma \text{pol-upd-triv2}: t k = \lfloor \text{deny } x \rfloor \implies t(k \mapsto_- x) = t
\text{ by (rule ext) simp}

lemma \text{pol-upd-allow-nonempty}: t(k \mapsto_+ x) \neq \emptyset
\text{ by simp}

lemma \text{pol-upd-deny-nonempty}: t(k \mapsto_- x) \neq \emptyset
\text{ by simp}

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

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

lemma \text{pol-upd-neq1 [simp]}: m(a \mapsto_+ x) \neq n(a \mapsto_- y)
\text{ 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).

\text{syntax}
- \text{-policyoverride} :: \[(a \mapsto_+ b, a \mapsto_- b) \implies a \mapsto_+ b \ (\text{infixl} \ 100)\]

\text{translations}
\[ p \oplus q \overset{\text{++}}{=} q \]

Some elementary facts inherited from Map are:

lemma \text{override-empty}: p \oplus \emptyset = p
\text{ by simp}

lemma \text{empty-override}: \emptyset \oplus p = p
\text{ by simp}

9
lemma override-associ: $p_1 \oplus (p_2 \oplus p_3) = (p_1 \oplus p_2) \oplus p_3$
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 ++'A 100)
where \(m_2 ++'A m_1 =
(\lambda x. (case\ m_1\ x\ of
[allow a] \Rightarrow [allow a]
| [deny a] \Rightarrow (case\ m_2\ x\ of\ [allow b] \Rightarrow [allow b]
| - \Rightarrow [deny a])
| \bot \Rightarrow m_2\ x))
\)

syntax
-policyoverride\_A :: \[''a \mapsto 'b, ''a \mapsto 'b\] \Rightarrow ''a \mapsto 'b (infixl \(\oplus' A 100\)
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

done

lemma override\_A-associ: \(p_1 \oplus' A (p_2 \oplus' A p_3) = (p_1 \oplus' A p_2) \oplus' A p_3\)
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 ++'D 100)
where \(m_1 ++'D m_2 =
(\lambda x. case\ m_2\ x\ of
[deny a] \Rightarrow [deny a]
| [allow a] \Rightarrow (case\ m_1\ x\ of\ [deny b] \Rightarrow [deny b])
\)
let · ⇒ · 

| · ⇒ · |
| · ⇒ · |

**syntax**

-policyoverride-D :: [a ↦ b, `a ↦ b] ⇒ `a ↦ b (infixl ⊔ D 100)

**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

**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 :: [β ⇒ γ, `α ⇒ β] ⇒ `α ⇒ γ (infixl o-f 55)

where

\( f \circ-f p = (\lambda x. \text{case } p x \text{ of} \)

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

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

\[ \bot \Rightarrow \bot \)

**syntax**

-policy-range-comp :: [β ⇒ γ, `α ⇒ β] ⇒ `α ⇒ γ (infixl o-f 55)

**translations**
\[ p \circ f \ q \iff p \circ f \ q \]

**Lemma** \( \text{policy-range-comp-strict} : f \circ \emptyset = \emptyset \)
- **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** \( \text{range-split} ::= [\{\beta \Rightarrow \gamma\} \times \{\beta \Rightarrow \gamma\}, \alpha \mapsto \beta] \Rightarrow \alpha \mapsto \gamma \)

**where** \((P) \nabla p = (\lambda x. \text{case } p x \text{ of})\)
- [allow \(y\)] \(\Rightarrow\) [allow ((fst \(P\)) \(y\))]
- [deny \(y\)] \(\Rightarrow\) [deny ((snd \(P\)) \(y\))]
- \(\bot\) \(\Rightarrow\) \(\bot\)

**Lemma** \( \text{range-split-strict}\): \(P \nabla \emptyset = \emptyset \)
- **apply** (rule ext)
- **apply** (simp add: range-split-def)
- **done**

**Lemma** \( \text{range-split-charn} \):
\((f,g) \nabla p = (\lambda x. \text{case } p x \text{ of})\)
- [allow \(x\)] \(\Rightarrow\) [allow \((f \ x)\)]
- [deny \(x\)] \(\Rightarrow\) [deny \((g \ x)\)]
- \(\bot\) \(\Rightarrow\) \(\bot\)

**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** \( \text{range-split-vs-range-compose} : (f,f) \nabla p = f \circ f \ p \)
- **by** (simp add: range-split-charn policy-range-comp-def)
lemma range-split-id [simp]: (id,id) ∇ 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) ∇ (g1,g2) ∇ p = (f1 o g1, f2 o g2) ∇ 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
done
done
done

definition dom-split2a :: [('α ⇀ 'γ) × ('α ⇀ 'γ), 'α ↦→ 'β] ⇒ 'α ↦→ 'γ (infixr Δa 100)
where P Δa p = (λx. case p x of
     [allow y] ⇒ [allow (the ((fst P) x))]
     [deny y] ⇒ [deny (the ((snd P) x))]
     ⊥ ⇒ ⊥)

definition dom-split2 :: [('α ⇒ 'γ) × ('α ⇒ 'γ), 'α ↦→ 'γ] ⇒ 'α ↦→ 'γ (infixr Δ 100)
where P Δ p = (λx. case p x of
     [allow y] ⇒ [allow ((fst P) x)]
     [deny y] ⇒ [deny ((snd P) x)]
     ⊥ ⇒ ⊥)

definition range-split2 :: [('α ⇒ 'γ) × ('α ⇒ 'γ), 'α ↦→ ('β × 'γ)] ⇒ 'α ↦→ ('β × 'γ) (infixr ∇2
where $P \triangledown 2 p = (\lambda x. \text{case } p x \text{ of }$
\[
  [\text{allow } y] \Rightarrow [\text{allow } (y, (\text{fst } P) x)]
  | [\text{deny } y] \Rightarrow [\text{deny } (y, (\text{snd } P) x)]
  | \bot \Rightarrow \bot)
\]

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 :: $(\times \sigma \mapsto \sigma) \Rightarrow (\times \sigma \mapsto \sigma)\ MON\ SE$
\[
\text{where policy2MON } p = (\lambda i \sigma. \text{case } p (i, \sigma) \text{ of }$
\[
  [\text{allow outs}, \sigma'] \Rightarrow [\text{allow outs}, \sigma']
  | [\text{deny outs}, \sigma'] \Rightarrow [\text{deny outs}, \sigma']
  | \bot \Rightarrow \bot)
\]

**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**

**Definition**

**Definition** deny-pfun :: $(\alpha \mapsto \beta) \Rightarrow (\alpha \mapsto \beta)\ \text{(AllD)}$
\[
\text{where deny-pfun } pf \equiv (\lambda x. \text{case } pf x \text{ of }$
\[
  [y] \Rightarrow [\text{deny } y]
  | \bot \Rightarrow \bot)
\]

**Definition**

allow-pfun :: $(\alpha \mapsto \beta) \Rightarrow (\alpha \mapsto \beta)\ \text{(AllA)}$

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
where
allow-pfun \( pf \equiv (\lambda x. \text{case } pf x \text{ of} \)
\[ y \Rightarrow [\text{allow } (y)] \]
\[ \bot \Rightarrow \bot \])

syntax
-allow-pfun :: \((\alpha \to \beta) \Rightarrow (\alpha \to \beta) (A_p)
translations
\( A_p f \equiv AllA f \)

syntax
-deny-pfun :: \((\alpha \to \beta) \Rightarrow (\alpha \to \beta) (D_p)
translations
\( D_p f \equiv AllD f \)

notation
\( \text{deny-pfun } \text{(binder } \forall D 10 \text{) and} \)
\( \text{allow-pfun } \text{(binder } \forall A 10 \text{)} \)

lemma \( AllD\text{-norm}[simp]: \text{deny-pfun } (id \circ (\lambda x. \lfloor x \rfloor)) = (\forall Dx. \lfloor x \rfloor) \)
by(\text{simp add:id-def comp-def})

lemma \( AllD\text{-norm2}[simp]: \text{deny-pfun } (\text{Some } \circ \text{id}) = (\forall Dx. \lfloor x \rfloor) \)
by(\text{simp add:id-def comp-def})

lemma \( AllA\text{-norm}[simp]: \text{allow-pfun } (id \circ \text{Some}) = (\forall Ax. \lfloor x \rfloor) \)
by(\text{simp add:id-def comp-def})

lemma \( AllA\text{-norm2}[simp]: \text{allow-pfun } (\text{Some } \circ \text{id}) = (\forall Ax. \lfloor x \rfloor) \)
by(\text{simp add:id-def comp-def})

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

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

lemma \( \text{neg-Allow-Deny: } pf \neq \emptyset \Rightarrow (\text{deny-pfun } pf) \neq (\text{allow-pfun } pf) \)
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 \( pf x = \bot \)
apply (auto)
done
done

2.2.2 Common Instances

definition allow-all-fun :: ('α ⇒ 'β) ⇒ ('α ⇒ 'β) (A_f)
  where allow-all-fun f = allow-pfun (Some o f)

definition deny-all-fun :: ('α ⇒ 'β) ⇒ ('α ⇒ 'β) (D_f)
  where deny-all-fun f ≡ deny-pfun (Some o f)

definition deny-all-id :: 'α ↦→ 'α (D_I)
  where deny-all-id ≡ deny-pfun (id o Some)

definition allow-all-id :: 'α ↦→ 'α (A_I)
  where allow-all-id ≡ allow-pfun (id o Some)

definition allow-all :: ('α → unit) (A_U)
  where allow-all p = ⌊allow ()⌋

definition deny-all :: ('α → unit) (D_U)
  where deny-all p = ⌊deny ()⌋

... and resulting properties:

lemma A_I ⊕ Map.empty = A_I
  by simp

lemma A_f f ⊕ Map.empty = A_f f
  by simp

lemma allow-pfun Map.empty = Map.empty
  apply (rule ext)
  apply (simp add: allow-pfun-def)
  done

lemma allow-left-cancel : dom pf = UNIV → (allow-pfun pf) ⊕ x = (allow-pfun pf)
  apply (rule ext)+
  apply (auto simp: allow-pfun-def option.splits)
\textbf{done}

**lemma** \texttt{deny-left-cancel} : \texttt{dom pf = UNIV \implies (deny-pfun pf) \bigoplus x = (deny-pfun pf)}

\textbf{apply} (\texttt{rule ext})+

\textbf{by} (\texttt{auto simp: deny-pfun-def option.splits})

\section{Domain, Range, and Restrictions}

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

\texttt{Map.dom_def} : \texttt{dom ?m = \{a. ?m a \neq \bot\}}

whereas range is just an abbreviation for image:

\texttt{abbreviation range :: \{('a \Rightarrow 'b) \Rightarrow 'b set\}}

where -- "of function" "range f == f ' UNIV"

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

- \texttt{Map.domD} \texttt{?a \in dom \?m \implies \exists \?b. \?m \?a = \{b\}}
- \texttt{Map.domI} \texttt{?m \?a = \{?b\} \implies \?a \in dom \?m}
- \texttt{Map.domIff} (\texttt{?a \in dom \?m}) = (\texttt{?m ?a \neq \bot})
- \texttt{Map.dom_const} \texttt{dom (\lambda x. [?f x]) = UNIV}
- \texttt{Map.dom_def} \texttt{dom \?m = \{a. \?m a \neq \bot\}}
- \texttt{Map.dom_empty} \texttt{dom \emptyset = \{\}}
- \texttt{Map.dom_eq_empty_conv} (\texttt{dom \?f = \{\}}) = (\texttt{?f = \emptyset})
- \texttt{Map.dom_eq_singleton_conv} (\texttt{dom \?f = \{?x\}}) = (\exists \?v. \?f = [?x \mapsto \?v])
- \texttt{Map.dom_fun_upd} \texttt{dom (?f(?x := ?y)) = (if \?y = \bot then dom \?f - \{?x\} else insert \?x (dom \?f))}
- \texttt{Map.dom_if} \texttt{dom (\lambda x. if \?P x then \?f x else \?g x) = dom \?f \cap \{x. \?P x\} \cup dom \?g \cap \{x. \neg \?P x\}}
- \texttt{Map.dom_map_add} \texttt{dom (?n \bigoplus ?m) = dom \?n \cup dom \?m}

However, some properties are specific to policy concepts:

**lemma** \texttt{sub-ran} : \texttt{ran \ P \subseteq Allow \cup Deny}

\textbf{apply} (\texttt{auto simp: Allow-def Deny-def ran-def full-SetCompr-eq[symmetric]})[\texttt{1}]

\textbf{subgoal for} \texttt{x a}
apply (case-tac x)
apply (simp-all)
done

lemma dom-allow-pfun [simp]: dom(allow-pfun f) = dom f
apply (auto simp: allow-pfun-def)
subgoal for x y
  apply (case-tac f x, simp-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-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-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(A f 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-denpy-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]
done
subgoal for xa a
  apply (rule-tac x=a in exI)
  apply (simp)
done
done

lemma ran-denpy-all: ran(D f 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.

- Map.dom_map_add dom (?n \oplus ?m) = dom ?n \cup dom ?m
- Map.inj_on_map_add_dom inj-on (?m' \oplus ?m) (dom ?m') = inj-on ?m' (dom ?m)
- Map.map_add_comm dom ?m1 \cap dom ?m2 = {} \implies ?m2 \oplus ?m1 = ?m2 \oplus ?m1
- Map.map_add_dom_app_simps(1) ?m \in dom ?l2 \implies (?l2 \oplus ?l1) ?m = ?l2 ?m
- Map.map_add_dom_app_simps(2) ?m \notin dom ?l1 \implies (?l2 \oplus ?l1) ?m = ?l2 ?m
- Map.map_add_dom_app_simps(3) ?m \notin dom ?l2 \implies (?l2 \oplus ?l1) ?m = ?l2 ?m
- Map.map_add_upd_left ?m \notin dom ?e2 \implies ?e2 \oplus ?e1(?m \mapsto ?u1) = (?e2 \oplus ?e1)(?m \mapsto ?u1)

The latter rule also applies to allow- and deny-override.
definition dom-restrict :: \([\alpha\, set, \alpha\to\beta] \Rightarrow \alpha\to\beta\) (infixr \(<55\))
where \(S \triangleleft p \equiv (\lambda x. \text{if } x \in S \text{ then } p \ x \text{ else } \bot)\)

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

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 :: \([\beta\, decision\ set] \Rightarrow \alpha\to\beta\) (infixr \(<55\))
where \(p \triangleright S \equiv (\lambda x. \text{if } (\text{the } (p \ x)) \in (\text{Some}\ S) \text{ then } p \ x \text{ else } \bot)\)

definition ran-restrict2 :: \([\beta\, decision\ set] \Rightarrow \alpha\to\beta\) (infixr \(<55\))
where \(p \triangleright\!\!\!\!2 S \equiv (\lambda x. \text{if } (\text{the } (p \ x)) \in (S) \text{ then } p \ x \text{ else } \bot)\)

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 \(x\ xa\ xb\))
apply (simp-all)
apply (metis inj-Some inj-image-mem-iff)
done
done
lemma ran-ran-restrict[simp] : ran(p ▷ S) = S ∩ ran p
by (auto simp: ran-restrict-def image-def ran-def)

lemma ran-restrict-idem[simp] : p ▷ (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 ▷ S) ▷ T = p ▷ T ∩ 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] : (∀ Ax. ⌊P x⌋) ▷ Allow = (∀ Ax. ⌊P x⌋)
apply (rule ext)
apply (auto simp: Allow-def ran-restrict-def)
done

lemma ran-gen-D[simp] : (∀ Dx. ⌊P x⌋) ▷ Deny = (∀ Dx. ⌊P x⌋)
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 ⇒ ('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 ⇒ x = deny(deny y)
  apply (case-tac x)
  apply (rename-tac α)
  apply (case-tac α, simp-all)[1]
  done

lemma flat-orA-allow[dest]: flat-orA x = allow y ⇒ x = allow(allow y)
  ∨ x = allow(deny y)
  ∨ x = deny(allow y)
  apply (case-tac x)
  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]
  done

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

∀ x = \text{allow}(\text{deny} \ y)\\ \lor x = \text{deny}(\text{allow} \ y)

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

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

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

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

fun flat-2 :: ('α decision) decision ⇒ ('α decision)
where flat-2(\text{allow}(\text{allow} \ y)) = \text{allow} \ y
     | flat-2(\text{allow}(\text{deny} \ y)) = \text{deny} \ y
     | flat-2(\text{deny}(\text{allow} \ y)) = \text{allow} \ y
     | flat-2(\text{deny}(\text{deny} \ y)) = \text{deny} \ y

lemma flat-2-allow[dest]: flat-2 \ x = \text{allow} \ y ⇒ x = \text{allow}(\text{allow} \ y) \lor x = \text{deny}(\text{allow} \ y)
 apply (\text{case-tac} \ x)
apply (\text{rename-tac} \ α)
apply (\text{case-tac} \ α, \text{simp-all})[1]
apply (rename-tac α)
apply (case-tac α, simp-all)[1]
done

lemma flat-2-deny[dest]: flat-2 x = deny y ⇒ x = deny(deny 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

2.3.2 Policy Composition

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

fun lift :: ('α ⇒→ 'β) ⇒ ('α decision ⇒→ 'β decision)
where lift f (deny s) = (case f s of
  ⌊y⌋ ⇒⌊deny y⌋ |
  ⊥⇒⊥)
| lift f (allow s) = (case f s of
  ⌊y⌋ ⇒⌊allow y⌋ |
  ⊥⇒⊥)

lemma lift-mt [simp]: lift ∅ = ∅
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 ◦m p1)

notation comp-orA (infixl ◦o'orA 55)

lemma comp-orA-mt[simp]:p ◦o'orA ∅ = ∅
\begin{verbatim}
by (simp add: comp-orA-def)

lemma mt-comp-or\{simp\}:∅ o_{\land A} p = ∅
  by (simp add: comp-orA-def)

definition
  comp-orD :: [\beta \Rightarrow \gamma, \alpha \Rightarrow \beta] \Rightarrow \alpha \Rightarrow \gamma
    (infixl o'_orD 55) where
  p2 o-orD p1 ≡ (map-option flat-orD) o (lift p2 o_m p1)

notation
  comp-orD (infixl o_rD 55)

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

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

definition
  comp-1 :: [\beta \Rightarrow \gamma, \alpha \Rightarrow \beta] \Rightarrow \alpha \Rightarrow \gamma
    (infixl o'_1 55) where
  p2 o-1 p1 ≡ (map-option flat-1) o (lift p2 o_m p1)

notation
  comp-1 (infixl o_1 55)

lemma comp-1-mt[simp]:p o_1 ∅ = ∅
  by (simp add: comp-1-def)

lemma mt-comp-1[simp]:∅ o_1 p = ∅
  by (simp add: comp-1-def)

definition
  comp-2 :: [\beta \Rightarrow \gamma, \alpha \Rightarrow \beta] \Rightarrow \alpha \Rightarrow \gamma
    (infixl o'_2 55) where
  p2 o-2 p1 ≡ (map-option flat-2) o (lift p2 o_m p1)

notation
  comp-2 (infixl o_2 55)

lemma comp-2-mt[simp]:p o_2 ∅ = ∅
  by (simp add: comp-2-def)

lemma mt-comp-2[simp]:∅ o_2 p = ∅
  by (simp add: comp-2-def)
\end{verbatim}
2.4 Parallel Composition

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 \( \nabla \), decide which result argument is chosen; this might be the first or the latter or, in case that \( \beta = \gamma \), and \( \beta \) 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** \( \text{prod-orA} : [\alpha \mapsto \beta, \gamma \mapsto \delta] \Rightarrow (\alpha \times \gamma \mapsto \beta \times \delta) \) (infixr \( \otimes \)

**where** \( p1 \otimes \gamma \ A \ p2 = (\lambda (x,y). \ (\text{case } p1 \ x \ of \ [\text{allow } d1] \Rightarrow (\text{case } p2 \ y \ of \ [\text{allow } d2] \Rightarrow [\text{allow}(d1,d2)] \ | \ [\text{deny } d2] \Rightarrow [\text{allow}(d1,d2)] \ | \ \bot \Rightarrow \bot) \ | \ [\text{deny } d1] \Rightarrow (\text{case } p2 \ y \ of \ [\text{allow } d2] \Rightarrow [\text{allow}(d1,d2)] \ | \ [\text{deny } d2] \Rightarrow [\text{deny}(d1,d2)] \ | \ \bot \Rightarrow \bot) \ | \ \bot \Rightarrow \bot)) \)

**lemma** \( \text{prod-orA-mt}\ | \text{simp} : p \otimes \gamma \ A \ \emptyset = \emptyset \)

**apply** (rule ext)

**apply** (simp add: prod-orA-def)

**apply** (auto)

**apply** (simp split: option.splits decision.splits)
lemma mt-prod-orA[simp]:\(\emptyset \otimes_A p = \emptyset\)
apply (rule ext)
apply (simp add: prod-A-def)
done

lemma prod-orA-quasi-commute: \(p2 \otimes_A p1 = (((\lambda(x,y). (y,x)) \ o-f (p1 \otimes_A p2)))\)
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 \(\otimes_D\) 55)
where p1 \(\otimes_D\) 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]
| \_ \Rightarrow \_)
| deny d1 \Rightarrow (case p2 y of
| allow d2 \Rightarrow [deny d1,d2]
| deny d2 \Rightarrow [deny d1,d2]
| \_ \Rightarrow \_)
| \_ \Rightarrow \_))

lemma prod-orD-mt[simp]:\(p \otimes_D \emptyset = \emptyset\)
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]:\(\emptyset \otimes_D p = \emptyset\)
apply (rule ext)
apply (simp add: prod-D-def)
done

lemma prod-orD-quasi-commute: \(p2 \otimes_D p1 = (((\lambda(x,y). (y,x)) \ o-f (p1 \otimes_D p2)))\)
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 :: \[\alpha \to \beta, \gamma \mapsto \delta\] \Rightarrow (\alpha \times \gamma \mapsto \beta \times \delta) (\text{infixr} \times_1 55)**

where \( p_1 \times_1 p_2 \equiv \)

\((\lambda(x,y). \ (\text{case } p_1 x \text{ of }
\text{allow } d_1 \Rightarrow \text{(case } p_2 y \text{ of }
\text{allow } d_2 \Rightarrow \text{allow}(d_1,d_2)\}
\text{deny } d_2 \Rightarrow \text{allow}(d_1,d_2)\}
\text{\(\bot\Rightarrow \bot\))
\text{deny } d_1 \Rightarrow \text{(case } p_2 y \text{ of }
\text{allow } d_2 \Rightarrow \text{deny}(d_1,d_2)\}
\text{deny } d_2 \Rightarrow \text{deny}(d_1,d_2)\}
\text{\(\bot\Rightarrow \bot\))
\text{\(\bot\Rightarrow \bot\))})

**lemma prod-1-mt[simp]: p \times_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 \times_1 p = \emptyset**
apply (rule ext)
apply (simp add: prod-1-def)
done

**definition prod-2 :: \[\alpha \to \beta, \gamma \mapsto \delta\] \Rightarrow (\alpha \times \gamma \mapsto \beta \times \delta) (\text{infixr} \times_2 55)**

where \( p_1 \times_2 p_2 \equiv \)

\((\lambda(x,y). \ (\text{case } p_1 x \text{ of }
\text{allow } d_1 \Rightarrow \text{(case } p_2 y \text{ of }
\text{allow } d_2 \Rightarrow \text{allow}(d_1,d_2)\}
\text{deny } d_2 \Rightarrow \text{deny}(d_1,d_2)\}
\text{\(\bot\Rightarrow \bot\))
\text{deny } d_1 \Rightarrow \text{(case } p_2 y \text{ of }
\text{allow } d_2 \Rightarrow \text{allow}(d_1,d_2)\}
\text{deny } d_2 \Rightarrow \text{deny}(d_1,d_2)\}
\text{\(\bot\Rightarrow \bot\))
\text{\(\bot\Rightarrow \bot\))})

**lemma prod-2-mt[simp]: p \times_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 \(\otimes_1\) 55)
where \(p \otimes_1 q = (p \otimes_1 q) \circ (\lambda x. (x,x))\)

lemma prod-1-id-mt[simp]:\(p \otimes_1 \emptyset = \emptyset\)
apply (rule ext)
apply (simp add: prod-1-id-def)
done

lemma mt-prod-1-id[simp]:\(\emptyset \otimes_1 p = \emptyset\)
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 \(\otimes_2\) 55)
where \(p \otimes_2 q = (p \otimes_2 q) \circ (\lambda x. (x,x))\)

lemma prod-2-id-mt[simp]:\(p \otimes_2 \emptyset = \emptyset\)
apply (rule ext)
apply (simp add: prod-2-id-def)
done

lemma mt-prod-2-id[simp]:\(\emptyset \otimes_2 p = \emptyset\)
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 :: \((\alpha \to \beta) \Rightarrow (\delta \to \gamma) \Rightarrow (\alpha \times \delta \to \beta \times \gamma)\) (infixr \(\otimes\) 60)
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.

\[
\text{definition } \text{comp-ran-split} :: \left( i \times \sigma \twoheadrightarrow \sigma \right) \Rightarrow \left( i \times \sigma \times \sigma' \twoheadrightarrow \sigma \times \sigma' \right) \quad (\text{infixr } \otimes \nabla 100)
\]

where

\[
\begin{align*}
\text{where } P \otimes \nabla p & \equiv \lambda x. \text{case } p (\text{fst } x) \text{ of} \\
& \quad | \text{allow } y \Rightarrow \text{case } (\text{fst } P) (\text{snd } x) \text{ of } \bot \Rightarrow \bot | [z] \Rightarrow [\text{allow } (y,z)] \\
& \quad | \text{deny } y \Rightarrow \text{case } (\text{snd } P) (\text{snd } x) \text{ of } \bot \Rightarrow \bot | [z] \Rightarrow [\text{deny } (y,z)] \\
& \quad | \bot \Rightarrow \bot
\end{align*}
\]

An alternative characterisation of the operator is as follows:

\[
\text{lemma } \text{comp-ran-split-charn:}
\]

\[
(f, g) \otimes \nabla p = (\\
\begin{aligned}
((p \triangleright \text{Allow}) \otimes \vee A (A_p f)) & \sqcup \\
((p \triangleright \text{Deny}) \otimes \vee A (D_p g)))
\end{aligned}
\]

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

\[
\text{lemma } \text{distr-or1-a: } (F = F1 \bigoplus F2) \Rightarrow ((N \otimes F) o f) =
((N \otimes F1) o f) \bigoplus ((N \otimes F2) o f))
\]

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: \( (F = F1 \oplus F2) \implies ((g \circ f \circ f) =
  ((g \circ f ((N \otimes_1 F) \circ f)) \oplus (g \circ f ((N \otimes_1 F2) \circ f)))\)
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: \( (F = F1 \oplus F2) \implies (((N \otimes_2 F) \circ f) =
  (((N \otimes_2 F1) \circ f) \oplus ((N \otimes_2 F2) \circ f)))\)
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-id-def map-add-def
    split: decision.splits option.splits)
done
done

lemma distr-or2: \( (F = F1 \oplus F2) \implies ((r \circ f \circ f) =
  ((r \circ f ((N \otimes_2 F) \circ f)) \oplus (r \circ f ((N \otimes_2 F2) \circ f)))\)
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: \( (F = F1 \oplus F2) \implies ((g \circ f ((N \otimes \bigvee_A F) \circ f)) =
  ((g \circ f ((N \otimes_1 F) \circ f)) \oplus (g \circ f ((N \otimes_1 F2) \circ f)))\)
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-orA: \( (F = F1 \oplus F2) \implies ((g \circ f ((N \otimes \bigvee_A F) \circ f)) =
  ((g \circ f ((N \otimes_1 F) \circ f)) \oplus (g \circ f ((N \otimes_1 F2) \circ f)))\)
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
\[(g \circ f \circ ((N \otimes_A F_1) \circ f)) \oplus (g \circ f \circ ((N \otimes_A F_2) \circ f)))\]

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

done

lemma distr-orD: \((F = F_1 \oplus F_2) \implies ((g \circ f \circ ((N \otimes_D F) \circ f)) =
((g \circ f \circ ((N \otimes_D F_1) \circ f)) \oplus (g \circ f \circ ((N \otimes_D F_2) \circ f)))\)
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

done

lemma coerc-assoc: \((r \circ f P) \circ d = r \circ f (P \circ d)\)
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
parallel-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 ⇒ 'b) ⇒ bool
where  gap-free p = (dom p = UNIV)

Comparing Policies

Policy p is more defined than q:
definition more-defined :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ bool
where  more-defined p q = (dom q ⊆ dom p)

definition strictly-more-defined :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ bool
where  strictly-more-defined p q = (dom q ⊂ dom p)

lemma strictly-more-vs-more: strictly-more-defined p q ⇒ 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 ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ bool (infixl ⊑ 60)
where  p ⊑ A q = (∀ x. (case q x of ⌊allow y⌋ ⇒ (∃ z. p x = ⌊allow z⌋))
  | ⌊deny y⌋ ⇒ True
  | ⊥ ⇒ True)

lemma more-permissive-refl : p ⊑ A p
  unfolding more-permissive-def
  by(auto split : option.split decision.split)

  unfolding more-permissive-def
  apply(auto split : option.split decision.split)

subgoal for x y
  apply(erule-tac x = x and
         P = λx. case p'' x of ⊥ ⇒ True
                                  | ⌊allow y⌋ ⇒ (∃ z. p' x = ⌊allow z⌋)
                                  | ⌊deny y⌋ ⇒ 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 ⇒ b) ⇒ ('a ⇒ b) ⇒ bool (infixl ⊑ 60)
where p ⊑ q = (∀ x. (case q x of ⌊deny y⌋ ⇒ (∃ z. (p x = ⌊deny z⌋))
| ⌊allow y⌋ ⇒ True
| ⊥ ⇒ True))

lemma more-rejective-trans : p ⊑ p' ⊑ p'' ⇒ p ⊑ p''
unfolding more-rejective-def
apply (auto split : option.split decision.split)
subgoal for x y
apply (erule-tac x = x and
P = λx. case p'' x of ⊥ ⇒ True
| ⌊allow y⌋ ⇒ True
| ⌊deny y⌋ ⇒ (∃ z. p' x = ⌊deny z⌋ in allE)
apply (simp, elim exE)
by (erule-tac x = x in allE, simp)
done

lemma more-rejective-refl : p ⊑ p
unfolding more-rejective-def
by (auto split : option.split decision.split)

lemma A_I ⊑ A p
unfolding more-permissive-def allow-all-fun-def allow-pfun-def
by (auto split : option.split decision.split)

lemma A f f ⊑ 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 ::
('a ⇒ b) ⇒ ('a ⇒ a) ⇒ ('b ⇒ b) ⇒ ('a ⇒ b) ⇒ bool
(- ⊑ - - [50,50,50,50] 50)
where p ⊑ a' a b q ≡
(∀ a. case p a of ⊥ ⇒ True
| ⌊allow y⌋ ⇒ (∃ a' ∈ {x. a' = a}. b' = q a' 
∧ a' ∈ {x. a' = a}.
| ⌊deny y⌋ ⇒ (∃ a' ∈ {x. a' = a}. b' = q a' 
∧ a' ∈ {x. a' = a}).

34
\[ \exists b'. q a' = [\text{deny } b'] \
\land \text{abs}_b b' = y \] 

**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**:

- \( A: p \sqsubseteq_{f, g} p' \)
- \( B: p' \sqsubseteq_{f', g'} p'' \)

**Shows**:

\( p \sqsubseteq_{f o f', g o g'} p'' \)

**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]

**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]

**Done**

**Done**

### 2.5.3 Equivalence of Policies

**Equivalence over domain D**

**Definition**: `p-eq-dom` :: \( ('a \mapsto 'b) \Rightarrow ('a \mapsto 'b) \Rightarrow \text{bool} \)

**Where**:

\( p \approx_{D} q = (\forall x \in D. p x = q x) \)

**Definition**: `no-conflicts` :: \( ('a \mapsto 'b) \Rightarrow ('a \mapsto 'b) \Rightarrow \text{bool} \)

**Where**:

\( \text{no-conflicts } p q = (\text{dom } p = \text{dom } q \land (\forall x \in (\text{dom } p). \text{case } p x \text{ of } [\text{allow } y] \Rightarrow (\exists z. q x = [\text{allow } z]) \)

\[ | [\text{deny } y] \Rightarrow (\exists z. q x = [\text{deny } z]) \)\]

**Lemma**: `policy-eq`:

**Assumes**:

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

Miscellaneous

lemma dom-inter: [[dom p ∩ dom q = {y}]; p x = ⌊y⌋] ⇒ q x = ⊥
  by (auto)

lemma dom-eq: dom p ∩ dom q = {} ⇒ p ⊕ A q = p ⊕ D q
  unfolding override-A-def override-D-def
  by (rule ext, auto simp: dom-def split: prod.splits option splits decision.splits)

lemma dom-split-alt-def : ((f, g) ∆ p = (dom(p ⊞ Allow) ▷ (A f)) ⊕ (dom(p ⊞ Deny)
  ▷ (D f g)))
  apply (rule ext)
  apply (simp add: dom-split2-def Allow-def Deny-def dom-restrict-def
    deny-all-fun-def allow-all-fun-def map-add-def)
  apply (simp split: option.splits decision.splits)
  apply (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 ⇒ 'b) list ⇒ ('a ⇒ 'b) where
`list2policy` `l` = foldr (λ `x` `y`. `(x ⊕ y)`) `l` ∅

Determine the position of element of a list.

**fun** `position` :: 'α ⇒ 'α list ⇒ 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` ∈ 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 \(\Rightarrow\) p \(\neq\) [] \(\Rightarrow\) 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 \(\Rightarrow\) (rm-MT-rules C p) = p
apply (induct p)
apply (simp-all)
done

**lemma nMTcharn:** none-MT-rules C p = (\(\forall\) r \(\in\) set p. dom (C r) \(\neq\) {}) 
apply (induct p)
apply (simp-all)
done

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

**lemma notMTnMT:** [ a \(\in\) set p; none-MT-rules C p] \(\Rightarrow\) dom (C a) \(\neq\) {}
apply (simp add: nMTcharn)
done

**lemma none-MT-rulesconc:** none-MT-rules C (a@[b]) \(\Rightarrow\) none-MT-rules C a
apply (induct a)
apply (simp-all)
done

**lemma nMTtail:** none-MT-rules C p \(\Rightarrow\) none-MT-rules C (tl p)
apply (induct p)
apply (simp-all)
done

**lemma not-MTimpnotMT[simp]:** not-MT C p \(\Rightarrow\) p \(\neq\) []
apply (auto)
done
lemma SR$n$MT: $\neg$ not-MT $C$ $p \implies$ rm-MT-rules $C$ $p = []$
apply (induct $p$
apply (auto simp: if-splits)
done

lemma NMPcharn: $[a \in \text{set } p; \text{dom } (C a) \neq \{\}] \implies$ not-MT $C$ $p$
apply (induct $p$
apply (auto simp: if-splits)
done

lemma NMPrrm: 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: [applied-rule-rev $C$ $x$ $b$ = Some $r$; applied-rule-rev $C$ $x$ $c$ = Some $r$] $\implies$
  applied-rule-rev $C$ $x$ $(a \# b)$ = applied-rule-rev $C$ $x$ $(a \# c)$
by (simp add: mrconc)

lemma mrconcNone: applied-rule-rev $C$ $x$ $p$ = None $\implies$
  applied-rule-rev $C$ $x$ $(b \# p)$ = applied-rule-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 $x \in \text{dom } (C ys)$)
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: [applied-rule-rev C x b = None; applied-rule-rev C x c = None] 
  \Rightarrow
  applied-rule-rev C x (a\#b) = applied-rule-rev C x (a\#c)
by (metis mrconcNone)

lemma mreq-end2: applied-rule-rev C x b = applied-rule-rev C x c 
  \Rightarrow
  applied-rule-rev C x (a\#b) = applied-rule-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-rule-rev C x p \neq None 
  \Rightarrow
  applied-rule-rev C x (b \# p) = applied-rule-rev C x (p)
by (auto simp: mrconc)

lemma mrNoneMT: [r \in set p; applied-rule-rev C x p = None] 
  \Rightarrow
  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 \in 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.

```plaintext
fun prod-list :: (\alpha \mapsto \beta) \Rightarrow ((\gamma \mapsto \delta) \text{ list}) \Rightarrow
(\alpha \mapsto \beta) \Rightarrow (\gamma \mapsto \delta) \Rightarrow ((\alpha \times \gamma) \mapsto (\beta \times \delta)) \Rightarrow
((\beta \times \delta) \Rightarrow \gamma \Rightarrow (x \Rightarrow (\alpha \times \gamma)) \Rightarrow
((x \mapsto \gamma) \text{ list}) \text{ (infixr } \otimes L 54) \text{ where }
prod-list x (y#ys) \text{ par-comb ran-adapt dom-adapt } =
((\text{ran-adapt o-f ((par-comb x y) o dom-adapt)}) # (\text{prod-list} x ys \text{ par-comb ran-adapt dom-adapt}))
| prod-list x [] \text{ par-comb ran-adapt dom-adapt } = []
```

An instance, as usual there are four of them.

```plaintext
definition prod-2-list :: [(\alpha \mapsto \beta), ((\gamma \mapsto \delta) \text{ list})] \Rightarrow
((\beta \times \delta) \Rightarrow \gamma \Rightarrow (x \Rightarrow (\alpha \times \gamma)) \Rightarrow
((x \mapsto \gamma) \text{ list}) \text{ (infixr } \otimes L 55) \text{ where }
x \otimes_2 y = (\lambda d r. (x \otimes_2 y) (\otimes_2) d r)
```

```plaintext
lemma list2listNMT: x \neq [] \Rightarrow map sem x \neq []
apply (case-tac x)
apply (simp-all)
done
```

```plaintext
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.

```plaintext
definition is-distr where
is-distr p = (\lambda g f. (\forall N P1 P2. ((g o-f ((p N (P1 \oplus P2)) o f)) =
((g o-f ((p N P1) o f)) \oplus (g o-f ((p N P2) o f)))))
```

```plaintext
definition is-strict where
is-strict p = (\lambda r d. \forall P1. (r o-f (p P1 \emptyset o d)) = \emptyset)
```

```plaintext
lemma is-distr-orD: is-distr (\otimes \lor D) d r
apply (simp add: is-distr-def)
apply (rule allI)+
apply (rule distr-orD)
apply (simp)
```

41
done

lemma is-strict-orD: is-strict \((\otimes \lor D) d r\)
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
done

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

lemma Norm-Distr-1: \(\big((r \circ f \circ (P1 \circ \text{list2policy } P2) \circ d)\big) x = ((\text{list2policy } ((P1 \otimes L P2) (\otimes 1) r d)) x)\)
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 \(x \in \text{dom } (r \circ f \circ (P1 \circ \text{list2policy } P2))\))
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)

42
lemma Norm-Distr-2: \((r \circ f (((\otimes 2) P1 \ (\text{list2policy} \ P2)) \circ d)) \ x = ((\text{list2policy} \ ((P1 \ \otimes L \ P2) \ (\otimes 2) \ r \ d)) \ x))\) 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 \(x \in \text{dom} \ ((r \circ f) \ ((P1 \ \otimes 2 \ p) \circ d)))\))
  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

lemma Norm-Distr-A: \((r \circ f (((\oplus A) P1 \ (\text{list2policy} \ P2)) \circ d)) \ x = ((\text{list2policy} \ ((P1 \ \otimes L \ P2) \ (\oplus A) \ r \ d)) \ x))\) 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 \(x \in \text{dom} \ ((r \circ f) \ ((P1 \ \otimes A \ p) \circ d)))\))
  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

lemma Norm-Distr-D: \((r \circ f (((\oplus D) P1 \ (\text{list2policy} \ P2)) \circ d)) \ x = ((\text{list2policy} \ ((P1 \ \otimes L \ P2) \ (\oplus D) \ r \ d)) \ x))\) 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 ⊗_D p) 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: dom A = UNIV ⇒ dom ((λ(x, y). x) o-f (A ⊗_1 B) o (λ x. (x,x))) = 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: dom A = UNIV ⇒ dom ((λ(x, y). x) o-f (A ⊗_2 B) o (λ x. (x,x))) = 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 ⇒ dom ((λ(x, y). x) o-f (A ⊗_A B) o (λ 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 ⇒ dom ((λ(x, y). x) o-f (A ⊗_D B) o (λ 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) = (∀y∈(set xs). dom x ∩ dom y = {}) ∧ disjDom xs
|disjDom [] = True

fun PUTList :: ('a =>→ 'b) ⇒ 'a ⇒ ('a =>→ 'b) list ⇒ bool
where
  PUTList PUT x (p#ps) = ((x ∈ dom p ⇒ (PUT x = p x)) ∧ (PUTList PUT x ps))
|PUTList PUT x [] = True

lemma distrPUTL1: x ∈ dom P ⇒ (list2policy PL) x = P x
  ⇒ (PUTList PUT x PL ⇒ (PUT x = P x))
  apply (induct PL)
  apply (auto simp: list2policy-def dom-def)
done

lemma PUTList-None: x /∈ dom (list2policy list) ⇒ PUTList PUT x list
  apply (induct list)
  apply (auto simp: list2policy-def dom-def)
done

lemma PUTList-DomMT:
  (∀y∈set list. dom a ∩ dom y = {}) ⇒ x ∈ (dom a) ⇒ x /∈ dom (list2policy list)
apply (induct list)
apply (auto simp: dom-def list2policy-def)
done

lemma distrPUTL2:
\[ x \in \text{dom } P \implies (\text{list2policy } PL \ x = P \ x \implies \text{disjDom } PL \implies (\text{PUT } x = P \ x) \implies \text{PUTList } \text{PUT } x \ PL \]
apply (induct PL)
apply (simp-all add: list2policy-def)
apply (auto)[1]
subgoal for a PL p
apply (case-tac x \in \text{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:
\[ [x \in \text{dom } P; (\text{list2policy } PL \ x = P \ x; \text{disjDom } PL] \implies (\text{PUT } x = P \ x) = \text{PUTList } \text{PUT } x \ PL \]
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
\[ \text{gatherDomain-aux } (x\#xs) = (\text{dom } x \cup (\text{gatherDomain-aux } xs)) \]
\[ \text{gatherDomain-aux } [] = \{\} \]

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

definition PUTListGD where PUTListGD PUT x p =
\[ (((x \notin (\text{gatherDomain } p) \land x \in \text{dom } (\text{last } p)) \implies \text{PUT } x = (\text{last } p) \ x) \land \]
\[ (\text{PUTList } \text{PUT } x \ (\text{butlast } p))) \]
definition \( \text{disjDomGD} \) where \( \text{disjDomGD} \ p = \text{disjDom} \ (\text{butlast} \ p) \)

lemma \( \text{distrPUTLG1} \): \[ x \in \text{dom} \ P; (\text{list2policy} \ PL) \ x = P \ x; \text{PUTListGD} \ PUT \ x \ PL \] \[ \Rightarrow \text{PUT} \ x = P \ x \]
\begin{align*}
\text{apply} & \ (\text{induct} \ PL) \\
\text{apply} & \ (\text{simp-all} \ \text{add: domIff \ PUTListGD-def \ disjDomGD-def \ gatherDomain-def \ list2policy-def}) \\
\text{apply} & \ (\text{auto} \ \text{simp: dom-def \ domIff \ split: if-split-asm}) \\
\text{done}
\end{align*}

lemma \( \text{distrPUTLG2} \):
\[ \text{PL} \neq [] \Rightarrow x \in \text{dom} \ P \Rightarrow (\text{list2policy} \ (PL)) \ x = P \ x \Rightarrow \text{disjDomGD} \ PL \Rightarrow \ (\text{PUT} \ x = P \ x) \Rightarrow \text{PUTListGD} \ \text{PUT} \ x \ (PL) \]
\begin{align*}
\text{apply} & \ (\text{simp \ add: PUTListGD-def \ disjDomGD-def \ gatherDomain-def \ list2policy-def}) \\
\text{apply} & \ (\text{induct} \ PL) \\
\text{apply} & \ (\text{auto}) \\
\text{apply} & \ (\text{metis \ PUTList-DomMT \ PUTList-None \ domI}) \\
\text{done}
\end{align*}

lemma \( \text{distrPUTLG} \):
\[ x \in \text{dom} \ P; (\text{list2policy} \ PL) \ x = P \ x; \text{disjDomGD} \ PL; \text{PL} \neq []; \]
\[ (\text{PUT} \ x = P \ x) = \text{PUTListGD} \ \text{PUT} \ x \ PL \]
\begin{align*}
\text{apply} & \ (\text{rule \ iffI}) \\
\text{apply} & \ (\text{rule \ distrPUTLG2}) \\
\text{apply} & \ (\text{simp-all}) \\
\text{apply} & \ (\text{rule-tac} \ PL = PL \ in \ distrPUTLG1) \\
\text{apply} & \ (\text{auto}) \\
\text{done}
\end{align*}

end

2.8 Putting Everything Together: UPF

theory \( \text{UPF} \)
\begin{align*}
\text{imports} & \ \text{Normalisation} \\
& \ \text{NormalisationTestSpecification} \\
& \ \text{Analysis}
\end{align*}
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 \( \text{UPFDefs} = \text{UPFCoreDefs ParallelDefs ElementaryPoliciesDefs} \)
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.
**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` | `Closed`

- **type-synonym** `entry-id` = `int`
- **type-synonym** `entry` = `status` × `user` × `data`
- **type-synonym** `SCR` = `(entry-id → entry)`
- **type-synonym** `DB` = `patient` → `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 SCROp :: (Operation × DB) → SCR where
| SCROp ((createSCR u r p), S) = S p
| SCROp ((appendEntry u r p ei e), S) = S p
| SCROp ((deleteEntry u r p e-id), S) = S p
| SCROp ((readEntry u r p e), S) = S p
| SCROp ((readSCR u r p), S) = S p
| SCROp ((changeStatus u r p s ei), S) = S p
| SCROp ((deleteSCR u r p), S) = S p
| SCROp ((editEntry u r p e-id s), S) = S p
| SCROp x = ⊥

fun patientOfOp :: Operation ⇒ patient where
| patientOfOp (createSCR u r p) = 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 e ei e) = (snd (snd e))
  contentOfOp (editEntry u r p e ei e) = (snd (snd e))

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

fun allContentStatic where
allContentStatic (x#xs) = (contentStatic x ∧ allContentStatic xs)
|allContentStatic [] = True

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 × DB → unit)

definition get-entry:: DB ⇒ patient ⇒ entry-id ⇒ entry option where
get-entry S p e-id = (case S p of ⊥ ⇒ ⊥
| [Scr] ⇒ Scr e-id)

definition userHasAccess:: user ⇒ entry ⇒ bool where
userHasAccess u e = ((fst e) = Open ∨ (fst (snd e) = u))

definition readEntrySE :: SEPolicy where
readEntrySE x = (case x of (readEntry u r p e-id,S) ⇒ (case get-entry S p e-id of
⊥ ⇒ ⊥
| [e] ⇒ (if (userHasAccess u e)
then [allow ()] 
else [deny ()] )))

| x ⇒ ⊥)

definition deleteEntrySE :: SEPolicy where
deleteEntrySE x = (case x of (deleteEntry u r p e-id,S) ⇒ (case get-entry S p e-id of
⊥ ⇒ ⊥
| [e] ⇒ (if (userHasAccess u e)
then [allow ()] 
else [deny ()] ))

| x ⇒ ⊥)

definition editEntrySE :: SEPolicy where
editEntrySE x = (case x of (editEntry u r p e-id s,S) ⇒ (case get-entry S p e-id of
⊥ ⇒ ⊥
| [e] ⇒ (if (userHasAccess u e)
then [allow ()] 
else [deny ()] )))
definition \textit{SEPolicy} :: SEPolicy where
\begin{align*}
\text{SEPolicy} &= \text{editEntrySE} \oplus \text{deleteEntrySE} \oplus \text{readEntrySE} \oplus A_U
\end{align*}

lemmas \textit{SEsimps} = \text{SEPolicy-def get-entry-def userHasAccess-def} \\
&\quad \text{editEntrySE-def deleteEntrySE-def readEntrySE-def}

Legitimate Relationships

\textbf{type-synonym} \textit{LRPolicy} = (\text{Operation} \times \Sigma, \text{unit}) \text{ policy}

fun \textit{hasLR} :: user \Rightarrow patient \Rightarrow \Sigma \Rightarrow \text{bool} where
\begin{align*}
\text{hasLR} \ u \ p \ \Sigma &= \begin{cases}
\text{False} & \text{if } \Sigma p \ \bot \\
(\exists \ lr. \ lr \in (\text{ran lrs}) \land u \in lr) & \text{otherwise}
\end{cases}
\end{align*}

\textbf{definition} \textit{LRPolicy} :: LRPolicy where
\begin{align*}
\text{LRPolicy} &= (\lambda (x,y). \ (\text{if } \text{hasLR} \ (\text{userOfOp x}) \ (\text{patientOfOp x}) \ y \\
&\quad \text{then } \allow (\ )) \\
&\quad \text{else } \deny (\ )))
\end{align*}

\textbf{definition} \textit{createSCRPolicy} :: LRPolicy where
\begin{align*}
\text{createSCRPolicy} \ x &= \begin{cases}
\allow (\ ) & \text{if } (\text{is-createSCR} \ (\text{fst x})) \\
\bot & \text{otherwise}
\end{cases}
\end{align*}

\textbf{definition} \textit{addLRPolicy} :: LRPolicy where
\begin{align*}
\text{addLRPolicy} \ x &= \begin{cases}
\allow (\ ) & \text{if } (\text{is-addLR} \ (\text{fst x})) \\
\bot & \text{otherwise}
\end{cases}
\end{align*}

\textbf{definition} \textit{LR-Policy} where \ \textit{LR-Policy} = \text{createSCRPolicy} \oplus \text{addLRPolicy} \oplus \text{LR-Policy} \oplus A_U

\textbf{lemmas} \textit{LRsimps} = \text{LR-Policy-def createSCRPolicy-def addLRPolicy-def LRPolicy-def}

\textbf{type-synonym} \textit{FunPolicy} = (\text{Operation} \times DB \times \Sigma, \text{unit}) \text{ policy}

fun \textit{createFunPolicy} :: FunPolicy where
\begin{align*}
\text{createFunPolicy} \ ((\text{createSCR} \ u \ p), (D,S)) &= \begin{cases}
\deny (\ ) & \text{if } p \in \text{dom D} \\
\allow (\ ) & \text{otherwise}
\end{cases}
\end{align*}

\begin{align*}
\text{createFunPolicy} \ x &= \bot
\end{align*}
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,(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
  then ⌊allow ()⌋
  else ⌊deny ()⌋)
  | - ⇒ ⌊deny ()⌋)
then |allow ()|
else |deny ()|
|readEntryFunPolicy x = ⊥

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

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

definition FunPolicy where
FunPolicy = editEntryFunPolicy ∘ appendEntryFunPolicy ∘ readEntryFunPolicy ∘ deleteEntryFunPolicy ∘ changeStatusFunPolicy ∘ deleteSCRFunPolicy ∘ addLRFunPolicy ∘ readSCRFunPolicy ∘ createFunPolicy ⊕ AU

Modelling Core RBAC

type-synonym RBACPolicy = Operation × υ ↦ → unit

definition RBAC :: (role × Operation) set where
RBAC = \{(r,f). r = Nurse ∧ is-readEntry f\} ∪
\{(r,f). r = Nurse ∧ is-readSCR f\} ∪
\{(r,f). r = ClinicalPractitioner ∧ is-appendEntry f\} ∪
\{(r,f). r = ClinicalPractitioner ∧ is-deleteEntry f\} ∪
\{(r,f). r = ClinicalPractitioner ∧ is-readEntry f\} ∪
\{(r,f). r = ClinicalPractitioner ∧ is-readSCR f\} ∪
\{(r,f). r = ClinicalPractitioner ∧ is-changeStatus f\} ∪
\{(r,f). r = ClinicalPractitioner ∧ is-editEntry f\} ∪
\{(r,f). r = Clerical ∧ is-createSCR f\} ∪
\{(r,f). r = Clerical ∧ is-deleteSCR f\} ∪
\{(r,f). r = Clerical ∧ is-addLR f\} ∪
{(r,f). r = Clerical ∧ is-removeLR f}

definition RBACPolicy :: RBACPolicy where
RBACPolicy = (λ (f,uc).
    if ((roleOfOp f,f) ∈ RBAC ∧ ⌊roleOfOp f⌋ = uc (userOfOp f))
    then ⌊allow ()⌋
    else ⌊deny ()⌋)

3.1.4 The State Transitions and Output Function

State Transition

fun OpSuccessDB :: (Operation × DB) ↦ DB where
OpSuccessDB (createSCR u r p,S) = (case S p of ⊥ ⇒ ⌊S(p→0)⌋
    | ⌊x⌋ ⇒ ⌊S⌋)
| OpSuccessDB ((appendEntry u r p ei e),S) =
    (case S p of ⊥ ⇒ ⌊S⌋
    | ⌊x⌋ ⇒ (if ei ∈ (dom x)
        then ⌊S⌋
        else ⌊S(p→x(ei→e))⌋))
| OpSuccessDB ((deleteSCR u r p),S) = (Some (S(p:=⊥)))
| OpSuccessDB ((deleteEntry u r p ei),S) =
    (case S p of ⊥⇒ ⌊S⌋
    | ⌊x⌋⇒ Some (S(p↦→(x(ei:=⊥)))))
| OpSuccessDB ((changeStatus u r p ei s),S) =
    (case S p of ⊥⇒ ⌊S⌋
    | ⌊x⌋⇒ (case x ei of
        ⌊e⌋⇒ ⌊S(p→x(ei→(s,snd e)))⌋
        | ⊥⇒ ⌊S⌋))
| OpSuccessDB ((editEntry u r p ei e),S) =
    (case S p of ⊥⇒ ⌊S⌋
    | ⌊x⌋⇒ (case x ei of
        ⌊e⌋⇒ ⌊S(p→x(ei→(e)))⌋
        | ⊥⇒ ⌊S⌋))
| OpSuccessDB (x,S) = ⌊S⌋

fun OpSuccessSigma :: (Operation × Σ) ↦ Σ where
OpSuccessSigma (addLR u r p lr-id us,S) =
    (case S p of ⌊lrs⌋ ⇒ (case (lrs lr-id) of
        ⊥⇒ ⌊S(p→(lrs(lr-id→us)))⌋
        | ⌊x⌋⇒ ⌊S⌋)
        | ⊥⇒ ⌊S(p→(Map.empty(lr-id→us)))⌋))
| OpSuccessSigma (removeLR u r p lr-id,S) =
\[(\text{case } S \ p \ \text{of } \text{Some} \ \text{lrs} \Rightarrow [S(p\rightarrow(lrs\text{-id}:=\bot))])\]\
\[| \bot \Rightarrow [S])\]
\[|\text{OpSuccessSigma } (x,S) = [S]\]

\text{fun } \text{OpSuccessUC} :: (\text{Operation} \times w) \rightarrow w \ \text{where} \quad \text{OpSuccessUC } (f,u) = [u]

\text{Output}

\text{type-synonym } \text{Output} = \text{unit}

\text{fun } \text{OpSuccessOutput} :: (\text{Operation}) \rightarrow \text{Output} \ \text{where} \quad \text{OpSuccessOutput } x = [()]\]

\text{fun } \text{OpFailOutput} :: \text{Operation} \rightarrow \text{Output} \ \text{where} \quad \text{OpFailOutput } x = [()]\]

3.1.5 Combine All Parts

\text{definition } \text{SE-LR-Policy} :: (\text{Operation} \times \text{DB} \times \Sigma, \text{unit}) \ \text{policy} \ \text{where} \quad \text{SE-LR-Policy} = (\lambda (x,x). x) \ o_f (\text{SEPolicy} \ \otimes_D \text{LR-Policy}) \ o (\lambda(a,b,c). ((a,b),a,c))

\text{definition } \text{SE-LR-FUN-Policy} :: (\text{Operation} \times \text{DB} \times \Sigma, \text{unit}) \ \text{policy} \ \text{where} \quad \text{SE-LR-FUN-Policy} = ((\lambda(x,x). x) \ o_f (\text{FunPolicy} \ \otimes_D \text{SE-LR-Policy}) \ o (\lambda a. (a,a)))

\text{definition } \text{SE-LR-RBAC-Policy} :: (\text{Operation} \times \text{DB} \times \Sigma \times w, \text{unit}) \ \text{policy} \ \text{where} \quad \text{SE-LR-RBAC-Policy} = (\lambda(x,x). x) \ o_f (\text{RBACPolicy} \ \otimes_D \text{SE-LR-FUN-Policy}) \ o (\lambda(a,b,c,d). ((a,d),(a,b,c)))

\text{definition } \text{ST-Allow} :: \text{Operation} \times \text{DB} \times \Sigma \times w \rightarrow \text{Output} \times \text{DB} \times \Sigma \times w \ \text{where} \quad \text{ST-Allow} = ((\text{OpSuccessOutput} \ \boxtimes_M (\text{OpSuccessDB} \ \boxtimes_S \text{OpSuccessSigma} \ \boxtimes_S \text{OpSuccessUC})) \ o (\lambda(a,b,c). ((a),(a,b,c))))\]

\text{definition } \text{ST-Deny} :: \text{Operation} \times \text{DB} \times \Sigma \times w \rightarrow \text{Output} \times \text{DB} \times \Sigma \times w \ \text{where} \quad \text{ST-Deny} = (\lambda (\text{ope},\text{sp},\text{si},\text{uc}). \text{Some } ((()), \text{sp},\text{si},\text{uc}))

\text{definition } \text{SE-LR-RBAC-ST-Policy} :: \text{Operation} \times \text{DB} \times \Sigma \times w \rightarrow \text{Output} \times \text{DB} \times \Sigma \times w
\[ \Sigma \times v \]

where \( SE-LR-RBAC-ST-Policy = ((\lambda (x,y).y) \circ ((ST-Allow,ST-Deny) \otimes \triangledown SE-LR-RBAC-Policy) \circ (\lambda x.(x,x))) \)

**definition** \( PolMon :: Operation \Rightarrow (Output \text{ decision},DB \times \Sigma \times v) \) \( MON_{SE} \)

**where** \( PolMon = (\text{policy2MON SE-LR-RBAC-ST-Policy}) \)

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** \( alice :: \text{user} \) **where** \( alice = 1 \)

**definition** \( bob :: \text{user} \) **where** \( bob = 2 \)

**definition** \( charlie :: \text{user} \) **where** \( charlie = 3 \)

**definition** \( patient1 :: \text{patient} \) **where** \( patient1 = 5 \)

**definition** \( patient2 :: \text{patient} \) **where** \( patient2 = 6 \)

**definition** \( UC0 :: v \) **where**

\( UC0 = \text{Map}.\text{empty}(alice\rightarrow\text{Nurse})(bob\rightarrow\text{ClinicalPractitioner})(charlie\rightarrow\text{Clerical}) \)

**definition** \( entry1 :: \text{entry} \) **where**

\( entry1 = (\text{Open},alice,\text{dummyContent}) \)

**definition** \( entry2 :: \text{entry} \) **where**

\( entry2 = (\text{Closed},bob,\text{dummyContent}) \)

**definition** \( entry3 :: \text{entry} \) **where**

\( entry3 = (\text{Closed},alice,\text{dummyContent}) \)

**definition** \( SCR1 :: SCR \) **where**

\( SCR1 = (\text{Map}.\text{empty}(1\rightarrow entry1)) \)
definition \( SCR2 :: SCR \) where
\[
SCR2 = (Map.\text{empty})
\]
definition \( Spine0 :: DB \) where
\[
Spine0 = Map.\text{empty}(\text{patient1} \mapsto SCR1)(\text{patient2} \mapsto SCR2)
\]
definition \( LR1 :: LR \) where
\[
LR1 = (Map.\text{empty}(1 \mapsto \{\text{alice}\}))
\]
definition \( \Sigma 0 :: \Sigma \) where
\[
\Sigma 0 = (Map.\text{empty}(\text{patient1} \mapsto LR1))
\]

### 3.2.2 The Initial System State

definition \( \sigma 0 :: DB \times \Sigma \times \upsilon \) where
\[
\sigma 0 = (Spine0,\Sigma 0,UC0)
\]

### 3.2.3 Basic Properties

lemma \([\text{simp}]\): \((\text{case } a \ of \ allow \ d \Rightarrow [X] \ |
\text{deny } d2 \Rightarrow [Y]) = \bot \implies False\)
by (case-tac a,simp-all)

lemma \([\text{cong},\text{simp}]\):
\((\text{if hasLR urp1-alice 1 } \Sigma 0 \text{ then [allow ()] else [deny ()]} = \bot) = False\)
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
RBACPolicy-def
\SE-LR-RBAC-ST-Policy-def map-add-def id-def \LRsimsps prod-2-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),σ0)= Some (deny ()
  by (simp add: PolSimps)

lemma exBool[simp]: ∃ a::bool. a
  by auto

lemma deny-allow[simp]: [deny ()] /∈ Some ' range allow
  by auto

lemma allow-deny[simp]: [allow ()] /∈ Some ' range deny
  by auto

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

lemma
  (σ0 |= (os ← mbind [(createSCR 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),σ0)= [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 \( (\sigma, 'o) \text{ MON}_{SE} = 'o \rightarrow (\sigma \times 'o) \)

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

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

notation bind-SE \( (\text{bind}_{SE}) \)

syntax
translations
\[ x \leftarrow f; g \Rightarrow \text{CONST bind-SE } f \ (% \ x \cdot g) \]

definition unit-SE :: 'a ⇒ ('a, 'σ)\text{MON}_{SE} \ ((\text{return } \cdot) \ 8)
where \ unit-SE \ e = (\lambda \sigma. \text{Some}(e, \sigma))
notation unit-SE (unit_{SE})

definition fail\_SE :: ('a, 'σ)\text{MON}_{SE}
where \ fail\_SE = (\lambda \sigma. \text{None})
notation fail\_SE (fail_{SE})

definition assert\_SE :: ('σ ⇒ bool) ⇒ (bool, 'a, 'σ)\text{MON}_{SE}
where \ assert\_SE \ P = (\lambda \sigma. \text{if } P \sigma \text{ then Some(True,} \sigma) \text{ else None})
notation assert\_SE (assert_{SE})

definition assume\_SE :: ('σ ⇒ bool) ⇒ (unit, 'a, 'σ)\text{MON}_{SE}
where \ assume\_SE \ P = (\lambda \sigma. \text{if } \exists \sigma . \ P \sigma \text{ then Some((), SOME } \sigma . \ P \sigma) \text{ else None})
notation assume\_SE (assume_{SE})

definition if-SE :: ['σ ⇒ bool, ('a, 'σ)\text{MON}_{SE}, ('a, 'σ)\text{MON}_{SE}] ⇒ ('a, 'σ)\text{MON}_{SE}
where \ if-SE \ c \ E \ F = (\lambda \sigma. \text{if } c \sigma \text{ then } E \sigma \text{ else } F \sigma)
notation if-SE (if_{SE})

The standard monad theorems about unit and associativity:

lemma bind-left-unit : (x ← return a; k) = k
  apply (simp add: unit-SE-def bind-SE-def)
  done

lemma bind-right-unit : (x ← m; return x) = m
  apply (simp add: unit-SE-def bind-SE-def)
  apply (rule ext)
  subgoal for σ
    apply (case-tac m σ)
    apply (simp-all)
    done
  done

lemma bind-assoc : (y ← (x ← m; k); h) = (x ← m; (y ← k; h))
  apply (simp add: unit-SE-def bind-SE-def)
  apply (rule ext)
  subgoal for σ
    apply (case-tac m σ, simp-all)
    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 \(op_1, op_2, \ldots, op_n\) with the inputs \(t_1, t_2, \ldots, t_n\) (outputs are treated analogously). Then we can encode for this interface the general input - type:

\[
\text{datatype in } = \ op_1 :: t_1 | \ldots | t_n
\]

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 \(op_1, op_2, \ldots, op_n\) with the inputs \(t_1, t_2, \ldots, t_n\) (outputs are treated analogously). Then we can encode for this interface the general input - type:

\[
\text{datatype in } = \ op_1 :: t_1 | \ldots | t_n
\]

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 ...
where \( \text{mbind} [] \ \text{iostep} \ \sigma = \text{Some}([], \sigma) \mid \)
\[
\text{mbind} (a \# H) \ \text{iostep} \ \sigma =
\begin{align*}
\text{(case iostep } a \ \text{of} \\
\text{None } & \Rightarrow \text{Some}([], \sigma) \\
| \text{Some } (out, \sigma') & \Rightarrow \text{(case mbind } H \ \text{iostep} \ \sigma' \ \text{of} \\
\text{None } & \Rightarrow \text{Some}([out], \sigma') \\
| \text{Some}(outs, \sigma'') & \Rightarrow \text{Some}(out \# outs, \sigma''))
\end{align*}
\]

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 \( \text{mbind}' \) defined below.

**Lemma** \( \text{mbind-unit} \) \([\text{simp}]\): \( \text{mbind} [] f = (\text{return } []) \)
by(\text{rule ext}, \text{simp add: unit-SE-def})

**Lemma** \( \text{mbind-nofailure} \) \([\text{simp}]\): \( \text{mbind} \ S \ f \ \sigma \neq \text{None} \)
apply (\text{rule-tac } x=\sigma \ \text{in spec})
apply (\text{induct } S)
using \( \text{mbind}.\text{simps}(1) \) apply force
apply(\text{simp add:unit-SE-def})
apply(\text{safe}[1])
subgoal for a S x
apply (\text{case-tac } f \ a \ x)
apply(\text{simp})
apply(\text{safe}[1])
subgoal for aa b
apply (\text{erule-tac } x=b \ \text{in allE})
apply (\text{erule exE}+)
apply (\text{simp})
done
done
done

The fail-strict version of \( \text{mbind}' \) looks as follows:

**fun** \( \text{mbind}' :: \ ' \ \text{list} \Rightarrow (' \Rightarrow ('o,'\sigma) \ MON_{SE}) \Rightarrow ('o \ \text{list}, \sigma) \ MON_{SE} \)

**where** \( \text{mbind}' [] \ \text{iostep} \ \sigma = \text{Some}([], \sigma) \mid \)
\[
\text{mbind}' (a \# H) \ \text{iostep} \ \sigma =
\begin{align*}
\text{(case iostep } a \ \text{of} \\
\text{None } & \Rightarrow \text{None} \\
| \text{Some } (out, \sigma') & \Rightarrow \text{(case mbind } H \ \text{iostep} \ \sigma' \ \text{of} \\
\text{None } & \Rightarrow \text{None} — \text{fail-strict} \\
| \text{Some}(outs, \sigma'') & \Rightarrow \text{Some}(out \# outs, \sigma''))
\end{align*}
\]

\( \text{mbind}' \) as failure strict operator can be seen as a foldr on bind—if the types would match ...
**Definition** \( \text{try-SE} :: (\prime o, \prime \sigma) \text{MON}_SE \Rightarrow (\prime o \text{ option}, \prime \sigma) \text{MON}_SE \)**

**Where**

\[
\text{try-SE} \ ioprog = (\lambda \sigma. \text{case } ioprog \ \sigma \ \text{of} \\
\quad \text{None } \Rightarrow \text{Some} \ (\text{None }, \sigma) \\
\quad \text{Some } (\text{outs }, \sigma') \Rightarrow \text{Some} \ (\text{Some } \text{outs }, \sigma')
\]

In contrast \( \text{mbind} \) as a failure safe operator can roughly be seen as a \( \text{foldr} \) on bind - \text{try}: \text{m1 ; try m2 ; try m3 ; ....} \. 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** \( \text{mbind-try}: \)

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

**Apply** (rule ext)

**Apply** (simp add: \( \text{bind-SE-def try-SE-def} \))

**Subgoal for** \( x \)

**Apply** (case-tac \( F \ a \ x \))

**Apply** (simp)

**Apply** (safe)[1]

**Apply** (simp add: \( \text{bind-SE-def try-SE-def} \))

**Subgoal for** \( aa \ b \)

**Apply** (case-tac \( \text{mbind} S F \ b \))

**Apply** (auto)

**Done**

**Done**

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

**Definition** \( \text{alt-SE} :: [(\prime o, \prime \sigma) \text{MON}_SE, (\prime o, \prime \sigma) \text{MON}_SE] \Rightarrow (\prime o, \prime \sigma) \text{MON}_SE \) \( (\text{infixl} \ \cap_{SE} 10) \)

**Where**

\[
(f \cap_{SE} g) = (\lambda \sigma. \text{case } \sigma \ \text{of} \text{ None } \Rightarrow g \ \sigma \\
\quad \text{Some } H \Rightarrow \text{Some } H)
\]

**Definition** \( \text{malt-SE} :: (\prime o, \prime \sigma) \text{MON}_SE \text{ list } \Rightarrow (\prime o, \prime \sigma) \text{MON}_SE \)

**Where** \( \text{malt-SE } S = \text{foldr} \ \text{alt-SE} S \ \text{fail}_{SE} \)

**Notation** \( \text{malt-SE } (\prod_{SE}) \)

**Lemma** \( \text{malt-SE-mt} [\text{simp}]: \prod_{SE} [] = \text{fail}_{SE} \)

**By** (simp add: \( \text{malt-SE-def} \))

**Lemma** \( \text{malt-SE-cons} [\text{simp}]: \prod_{SE} (a \ # \ S) = (a \cap_{SE} (\prod_{SE} S)) \)

69
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.

\[ \text{type-synonym} \ (\sigma, \sigma) \text{MON}_{SB} = \sigma \Rightarrow (\sigma \times \sigma) \text{set} \]

\[ \text{definition} \ \text{bind-SB} :: \ (\sigma, \sigma) \text{MON}_{SB} \Rightarrow (\sigma \Rightarrow (\sigma, \sigma) \text{MON}_{SB}) \Rightarrow (\sigma, \sigma) \text{MON}_{SB} \]

\[ \text{where} \ \text{bind-SB} \ f \ g \sigma = \bigcup \ ((\lambda(out, \sigma). (g out \sigma)) \cdot (f \sigma)) \]

\[ \text{notation} \ \text{bind-SB} \ (\text{bind}_{SB}) \]

\[ \text{definition} \ \text{unit-SB} :: \ \sigma \Rightarrow (\sigma, \sigma) \text{MON}_{SB} (((\text{returns}) \ 8) \Rightarrow 8) \]

\[ \text{where} \ \text{unit-SB} \ e = (\lambda\sigma. \{(e,\sigma)\}) \]

\[ \text{notation} \ \text{unit-SB} \ (\text{unit}_{SB}) \]

\[ \text{syntax} \ -\text{bind-SB} :: \ [\text{pttrn}, (\sigma, \sigma) \text{MON}_{SB}, (\sigma, \sigma) \text{MON}_{SB}] \Rightarrow (\sigma, \sigma) \text{MON}_{SB} \]

\[ \text{translations}\]

\[ x := f; \ g \Rightarrow \text{CONST} \ \text{bind-SB} \ f \ (\% x . \ g) \]

\[ \text{lemma} \ \text{bind-left-unit-SB} : \ (x := \text{returns} \ a; \ m) = m \]

\[ \text{apply} \ (\text{rule} \ \text{ext}) \]

\[ \text{apply} \ (\text{simp add} : \ \text{unit-SB-def} \ \text{bind-SB-def}) \]

\[ \text{done} \]

\[ \text{lemma} \ \text{bind-right-unit-SB} : \ (x := m; \ \text{returns} \ x) = m \]

\[ \text{apply} \ (\text{rule} \ \text{ext}) \]

\[ \text{apply} \ (\text{simp add} : \ \text{unit-SB-def} \ \text{bind-SB-def}) \]

\[ \text{done} \]

\[ \text{lemma} \ \text{bind-assoc-SB} : \ (y := (x := m; \ k); \ h) = (x := m; \ (y := k; \ h)) \]

\[ \text{apply} \ (\text{rule} \ \text{ext}) \]

\[ \text{apply} \ (\text{simp add} : \ \text{unit-SB-def} \ \text{bind-SB-def} \ \text{split-def}) \]

\[ \text{done} \]

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 \( (\alpha, \sigma) \text{MON}_{SBE} = \sigma \Rightarrow ((\alpha \times \sigma) \text{set}) \text{ option} \)

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

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

syntax -bind-SBE :: \[ \text{pttrn},(\alpha, \sigma)\text{MON}_{SBE},(\alpha', \sigma)\text{MON}_{SBE} \] \Rightarrow \( (\alpha', \sigma)\text{MON}_{SBE} \)

translations
\[ x \equiv f; \ g \equiv \text{CONST bind-SBE } f \ (\% x .\ g) \]

definition unit-SBE :: \( \alpha \Rightarrow (\alpha, \sigma)\text{MON}_{SBE} \) ((returning \-) \( \sigma \))

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

notation
\text{assert-SBE } (\text{assert}_{SBE})

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

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

notation
assert-SBE (\text{assert}_{SBE})

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

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

notation
assume-SBE (\text{assume}_{SBE})

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

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

notation
havoc-SBE (\text{havoc}_{SBE})

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

apply (rule ext)

apply (simp add: unit-SBE-def bind-SBE-def)

done

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

apply (rule ext)

apply (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.contr)

apply (simp add: Set.image-iff)

71
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 a b ab ba a b)
  then show ?case by (rule-tac x = (a, b) in bexI, simp-all)
next
  case (2 x a a b ab ba b)
  then show ?case by (rule-tac x = (a, b) in bexI, simp-all)
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 a b)
  then show ?case by (rule-tac x = (a, b) in bexI, simp-all)
next
  case (5 x a a b ab ba a b)
  then show ?case by (rule-tac x = (a, b) in bexI, simp-all)
next
  case (6 x a a b a b)
  then show ?case by (rule-tac x = (a, b) in bexI, simp-all)
qed
5.1.2 Valid Test Sequences in the State Exception Monad

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

definition valid-SE :: 'σ ⇒ (bool,'σ) MON_{SE} ⇒ bool (infix |= 15)
where (σ |= m) = (m σ ≠ None ∧ fst(the (m σ)))

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 :
σ |= (os ← (mbind ς ioprog); return(length ς = length os))
oops

lemma valid-true[simp]: (σ |= (s ← 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: ioprog a σ = None ⇒
(σ |= (s ← mbind (a#S) ioprog ; M s)) =
(σ |= (M []))
by(simp add: valid-SE-def unit-SE-def bind-SE-def)

lemma valid-failure': A σ = None ⇒ ¬(σ |= ((s ← A ; M s)))
by(simp add: valid-SE-def unit-SE-def bind-SE-def)

lemma valid-successElem:
M σ = Some(f σ,σ) ⇒ (σ |= M) = f σ
by(simp add: valid-SE-def unit-SE-def bind-SE-def )

lemma valid-success: ioprog a σ = Some(b,σ') ⇒
(σ |= (s ← mbind (a#S) ioprog ; M s)) =
(σ' |= (s ← mbind S ioprog ; M (b#s)))
apply (simp add: valid-SE-def unit-SE-def bind-SE-def )
apply (cases mbind S ioprog σ', auto)
done

lemma valid-success'': ioprog a σ = Some(b,σ') ⇒
(σ |= (s ← mbind (a#S) ioprog ; return (P s))) =
(σ' |= (s ← mbind S ioprog ; return (P (b#s))))
apply \(\text{simp add: valid-SE-def unit-SE-def bind-SE-def}\)
apply \(\text{cases mbind S ioprog }\sigma'\)
apply \(\text{simp-all}\)
apply \(\text{auto}\)
done

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

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

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

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

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

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

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

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

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

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

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

lemma assume-D : \( \sigma \models (x \leftarrow \text{assume}_{SE} P; M x)) \implies \exists \sigma. (P \sigma \land (\sigma \models (M ()))) \)
apply (auto simp: bind-SE-def assume-SE-def valid-SE-def split: HOL.if-split-asm)
apply (rule-tac x=Eps P in \text{exI})
apply (auto)[1]
subgoal for x a b
apply (rule-tac x=\text{True} in \text{exI}, rule-tac x=b in \text{exI})
apply (subst Hilbert-Choice.someI)
apply (assumption)
apply (simp)
done
apply (subst Hilbert-Choice.someI, assumption)
These two rules 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:**
\[
P \sigma = \implies (\sigma \mid SE P B_1 B_2) = (\sigma \mid B_1)
\]
by (auto simp: if-SE-def valid-SE-def)

**Lemma if-SE-D2:**
\[
\neg P \sigma = \implies (\sigma \mid SE P B_1 B_2) = (\sigma \mid B_2)
\]
by (auto simp: if-SE-def valid-SE-def)

**Lemma if-SE-split-asm:**
\[
(\sigma \mid SE P B_1 B_2) = ((P \sigma \land (\sigma \mid B_1)) \lor (\neg P \sigma \land (\sigma \mid B_2)))
\]
by (cases P \sigma, auto simp: if-SE-D1 if-SE-D2)

**Lemma if-SE-split:**
\[
(\sigma \mid SE P B_1 B_2) = ((P \sigma \implies (\sigma \mid B_1)) \land (\neg P \sigma \implies (\sigma \mid B_2)))
\]
by (cases P \sigma, auto simp: if-SE-D1 if-SE-D2)

**Lemma [code]:**
\[
(\sigma \mid m) = \text{case } (m \sigma) \text{ of None } \Rightarrow \text{False} | (\text{Some } (x,y)) \Rightarrow x
\]
apply (simp add: valid-SBE-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:**
\[
\sigma \Rightarrow (\mathrm{a,}'\sigma) \text{ MON}_{SBE} \Rightarrow \text{bool} \text{ (infix } \mid SBE 15)
\]
where \(\sigma \mid SBE m \equiv (m \sigma \neq \text{None})\)

This notation considers all non-failures as valid.

**Lemma assume-assert:**
\[
(\sigma \mid SBE (\_ \equiv \text{assume}_{SBE} P : \text{assert}_{SBE} Q)) = (P \sigma \implies Q \sigma)
\]
by (simp add: valid-SBE-def assume-SBE-def assert-SBE-def bind-SBE-def)

**Lemma assert-intro:**
\[
Q \sigma \implies \sigma \mid SBE (\text{assert}_{SBE} Q)
\]
by (simp add: valid-SBE-def assert-SBE-def bind-SBE-def)
end
Bibliography


