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
3.2 Instantiating Our Secure Service Example ........................................... 59
   3.2.1 Access Control Configuration .................................................. 59
   3.2.2 The Initial System State ....................................................... 60
   3.2.3 Basic Properties .................................................................. 60

4 Conclusion and Related Work ................................................................. 63
   4.1 Related Work ........................................................................ 63
   4.2 Conclusion Future Work ................................................................ 63

5 Appendix ............................................................................................ 65
   5.1 Basic Monad Theory for Sequential Computations ......................... 65
      5.1.1 General Framework for Monad-based Sequence-Test .............. 65
      5.1.2 Valid Test Sequences in the State Exception Monad ............. 73
      5.1.3 Valid Test Sequences in the State Exception Backtrack Monad .. 76
1 Introduction

Access control, i.e., restricting the access to information or resources, is an important pillar of today’s information security portfolio. Thus the large number of access control models (e.g., [1, 5, 6, 15–17, 19, 21]) and variants thereof (e.g., [2, 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 \( \beta \):

\[
\text{datatype} \quad \alpha \text{ decision} = \text{allow } \alpha \mid \text{deny } \alpha
\]

\[
\text{type-synonym} \quad (\alpha,\beta) \text{ policy} = \alpha \rightarrow \beta \text{ decision (infixr } \rightarrow)\)
\]

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

\[
\text{translations} \quad (\alpha \rightarrow) \beta \leq (\text{type}) \alpha \rightarrow \beta \text{ decision}
\]

\[
\text{type-notation} \quad \text{policy} (\text{infixr } \Rightarrow)\)
\]
... 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* ($\bot$)

**notation**  *Some* ([-, 80])

Thus, the range of a policy may consist of $[accept x]$ data, of $[deny x]$ 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 *Allow* and the deny-set *Deny* (written $A$ and $D$ set for short), to characterize these two main sets of the range of a policy.

**definition**  *Allow* :: (*'alpha decision*) set
**where**  *Allow* = range allow

**definition**  *Deny* :: (*'alpha decision*) set
**where**  *Deny* = range deny

**2.1.2 Policy Constructors**

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

Furthermore, we add notation adopted to our problem domain:

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

**syntax**

- *policylet1* :: ['a, 'a] => *policylet* (- /+/-)
- *policylet2* :: ['a, 'a] => *policylet* (- /-/-)
  :: *policylet* => *policylets* (-)
- *Maplets* :: *[policylet, policylets] => policylets* (-/-)
- *Maplets* :: *[policylet, policylets] => policylets* (-/-)
- *MapUpd* :: ['a |-+ 'b, policylets] => 'a |-> 'b (-/(-) [900,0][900])
- *emptypolicy* :: 'a |-> 'b (()</>

**translations**

- *MapUpd m (-Maplets xy ms) => -MapUpd (-MapUpd m xy) ms*
- *MapUpd m (-policylet1 x y) => m(x := CONST Some (CONST allow y))*
- *MapUpd m (-policylet2 x y) => m(x := CONST Some (CONST deny y))*
- $\emptyset$ => CONST Map.empty

Here are some lemmas essentially showing syntactic equivalences:

**lemma**  *test*: $\emptyset(x\mapsto a, y\mapsto b) = \emptyset(x\mapsto a, y\mapsto b)$  by simp
lemma test2: $p(x\mapsto +a, x\mapsto -b) = p(x\mapsto -b)$  \textbf{by} simp

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

lemma pol-upd-triv1: $t \; k = \allow x \implies t(k\mapsto +x) = t$
by (rule ext) simp

lemma pol-upd-triv2: $t \; k = \deny x \implies t(k\mapsto -x) = t$
by (rule ext) simp

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

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

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

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

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

2.1.3 Override Operators

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

syntax
-policyoverride :: ["a \mapsto 'b, 'a \mapsto 'b] => 'a \mapsto 'b (infixl \oplus 100)
translations
\[ p \oplus q \equiv q ++ p \]

Some elementary facts inherited from Map are:

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

lemma empty-override: $\emptyset \oplus p = p$
by simp
lemma override-assoc: \( p_1 \bigoplus_A (p_2 \bigoplus_A p_3) = (p_1 \bigoplus_A p_2) \bigoplus_A p_3 \)
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 :: \('[\alpha \mapsto \beta], ['\alpha \mapsto \beta] \Rightarrow ['\alpha \mapsto \beta]'\) (infix \(\bigoplus\) \(A\) \(100\))
where \( m_2 \bigoplus_A m_1 = \)
(λx. (case \( m_1\) \(x\) of
 \[ allow a \] ⇒ \[ allow a \]
| \[ deny a \] ⇒ (case \( m_2\) \(x\) of
 \[ allow b \] ⇒ \[ allow b \]
| \(=\) ⇒ \[ deny a \])
| \(\bot\) ⇒ \(m_2\) \(x\))
)

syntax
-policy-override-A :: \('[a \mapsto 'b], ['a \mapsto 'b] \Rightarrow ['a \mapsto 'b]'\) (infix \(\bigoplus\) \(A\) \(100\))
translations
\( p \bigoplus_A q \equiv p \bigoplus \neg A \) \(q\)

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

lemma empty-override-A[simp]: \( \emptyset \bigoplus_A p = p \)
apply (rule ext)
apply (simp add:override-A-def)
subgoal for \(x\)
apply (case-tac \(p\) \(x\))
apply (simp-all)
subgoal for \(a\)
apply (case-tac \(a\))
apply (simp-all)
done
done
done

lemma override-A-assoc: \( p_1 \bigoplus_A (p_2 \bigoplus_A p_3) = (p_1 \bigoplus_A p_2) \bigoplus_A p_3 \)
by (rule ext, simp add:override-A-def split: decision.splits option.splits)

definition override-D :: \('[\alpha \mapsto \beta], ['\alpha \mapsto \beta] \Rightarrow ['\alpha \mapsto \beta]'\) (infix \(\bigoplus\) \(D\) \(100\))
where \( m_1 \bigoplus_D m_2 = \)
(λx. (case \( m_2\) \(x\) of
 \[ deny a \] ⇒ \[ deny a \]
| \[ allow a \] ⇒ (case \( m_1\) \(x\) of \[ deny b \] ⇒ \[ deny b \])
)
allow a
⊥⇒ m1 x

syntax
-policyoverride-D :: [a ⇒ 'b, a ⇒ 'b] ⇒ a ⇒ 'b (infixl ⊕ D 100)
translations
p ⊕ D q ⇌ p ++-D q

lemma override-D-empty[simp]: p ⊕ D ⊥ = p
  by(simp add:override-D-def)

lemma empty-override-D[simp]: ⊥ ⊕ 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 ⊕ D (p2 ⊕ D p3) = (p1 ⊕ D p2) ⊕ 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 o-f p = (λx. case p x of
    [allow y] ⇒ [allow (f y)]
    | [deny y] ⇒ [deny (f y)]
    | ⊥ ⇒ ⊥)

syntax
-policy-range-comp :: '[β⇒ γ', 'α⇒'β] ⇒ 'α⇒'γ (infixl o′ 55)
translations
\[ p \circ f q \equiv p \circ f q \]

**Lemma** 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** range-split :: \((\beta' \Rightarrow \gamma') \times (\beta' \Rightarrow \gamma'), \alpha \mapsto \beta' \Rightarrow \gamma'\)

(infixr \(\nabla\) 100)

*where* \((P) \nabla p = (\lambda x. \text{case } p x \text{ of} \begin{array}{l}
\text{allow } y \Rightarrow \text{allow } ((\text{fst } P) y) \\
\text{deny } y \Rightarrow \text{deny } ((\text{snd } P) y) \\
\bot \Rightarrow \bot
\end{array}\)

**Lemma** range-split-strict\[\text{simp}\]: \(P \nabla \emptyset = \emptyset\)

*apply* (rule ext)

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

*done*

**Lemma** range-split-charn:

\((f,g) \nabla p = (\lambda x. \text{case } p x \text{ of} \begin{array}{l}
\text{allow } x \Rightarrow \text{allow } (f x) \\
\text{deny } x \Rightarrow \text{deny } (g x) \\
\bot \Rightarrow \bot
\end{array}\)

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

*apply* (rule ext)

*subgoal for* \(x\)

*apply* (case-tac \(p\) \(x\))

*apply* (simp-all)

*subgoal for* \(a\)

*apply* (case-tac \(a\))

*apply* (simp-all)

*done*

*done*

*done*

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

**Lemma** range-split-vs-range-compose: \((f,f) \nabla p = f \circ f p\)

*by* (simp add: range-split-charn policy-range-comp-def)
lemma range-split-id [simp]: \((id, id) \nabla p = p\)
apply (rule ext)
apply (simp add: range-split-charn id-def)

subgoal for \(x\)
apply (case-tac \(p x\))
apply (simp-all)

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

done
done

done

done

lemma range-split-bi-compose [simp]: \(((f1, f2) \nabla (g1, g2)) \nabla p = (f1 \circ g1 \circ f2 \circ g2) \nabla p\)
apply (rule ext)
apply (simp add: range-split-charn comp-def)

subgoal for \(x\)
apply (case-tac \(p x\))
apply (simp-all)

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

done
done

done
done

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

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

definition dom-split2a :: \([\alpha \rightarrow \gamma] \times [\alpha \rightarrow \gamma], \alpha \mapsto \beta \] \Rightarrow [\alpha \rightarrow \gamma] \quad \text{ (infixr } \Delta a 100)\)
where \(P \Delta a p = (\lambda x. \text{ case } p x \text{ of})\)
\(\begin{align*}
[\text{allow } y] & \Rightarrow [\text{allow } (\text{the } (\text{fst } P) x)] \\
[\text{deny } y] & \Rightarrow [\text{deny } (\text{the } (\text{snd } P) x)] \\
\bot & \Rightarrow \bot
\end{align*}\)

definition dom-split2 :: \([\alpha \Rightarrow \gamma] \times [\alpha \Rightarrow \gamma], \alpha \mapsto \beta \] \Rightarrow [\alpha \Rightarrow \gamma] \quad \text{ (infixr } \Delta 100)\)
where \(P \Delta p = (\lambda x. \text{ case } p x \text{ of})\)
\(\begin{align*}
[\text{allow } y] & \Rightarrow [\text{allow } (\text{the } (\text{fst } P) x)] \\
[\text{deny } y] & \Rightarrow [\text{deny } (\text{the } (\text{snd } P) x)] \\
\bot & \Rightarrow \bot
\end{align*}\)

definition range-split2 :: \([\alpha \Rightarrow \gamma] \times [\alpha \Rightarrow \gamma], \alpha \mapsto [\beta \times \gamma] \Rightarrow [\alpha \mapsto (\beta \times \gamma)] \quad \text{ (infixr } \nabla 2)\)
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 ::= (′ι × ′σ ⇒ → ′ι × ′σ) ⇒ (′ι ⇒ (′o decision,′σ) MONSE)

where policy2MON p = (λ ι σ. case p (ι,σ) of
⌊allow outs⌋ ⇒ ⌊allow outs⌋
| ⌊deny outs⌋ ⇒ ⌊deny outs⌋
| ⊥ ⇒ ⊥)

**lemmas** UPFCoreDefs = Allow-def Deny-def override-A-def override-D-def policy-range-comp-def range-split-def dom-split2-def map-add-def restrict-map-def

end

### 2.2 Elementary Policies

**theory** ElementaryPolicies

**imports** UPFCore

**begin**

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

#### 2.2.1 The Core Policy Combinators: Allow and Deny Everything

**definition** deny-pfun ::= (′α ⇒ ′β) ⇒ (′α ⇒ ′β) (AllD)

where deny-pfun pf ≡ (λ x. case pf x of
[y] ⇒ [deny (y)]
| ⊥ ⇒ ⊥)

**definition**

allow-pfun ::= (′α ⇒ ′β) ⇒ (′α ⇒ ′β) (AllA)
where
allow-pfun pf ≡ (λ x. case pf x of
    [y] ⇒ [ allow (y) ]
    |⊥⇒⊥)

syntax
-alow-pfun :: ('α →'β) ⇒ ('α → 'β) (A_p)
translations
A_p f ≜ AllA f

syntax
-deny-pfun :: ('α →'β) ⇒ ('α → 'β) (D_p)
translations
D_p f ≜ AllD f

notation
deny-pfun (binder ∀ D 10) and
allow-pfun (binder ∀ A 10)

lemma AllD-norm[simp]: deny-pfun (id o (λx. [x])) = (∀ Dx. [x])
by(simp add:id-def comp-def)

lemma AllD-norm2[simp]: deny-pfun (Some o id) = (∀ Dx. [x])
by(simp add:id-def comp-def)

lemma AllA-norm[simp]: allow-pfun (id o Some) = (∀ Ax. [x])
by(simp add:id-def comp-def)

lemma AllA-norm2[simp]: allow-pfun (Some o id) = (∀ Ax. [x])
by(simp add:id-def comp-def)

lemma AllA-apply[simp]: (∀ Ax. Some (P x)) x = [ allow (P x) ]
by(simp add:allow-pfun-def)

lemma AllD-apply[simp]: (∀ Dx. Some (P x)) x = [ deny (P x) ]
by(simp add:deny-pfun-def)

lemma neq-Allow-Deny: pf ≠ ∅ ⟹ (deny-pfun pf) ≠ (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 = ⊥)
2.2.2 Common Instances

definition allow-all-fun :: '(α ⇒ 'β) ⇒ ('α ⇒ 'β) (A₁)
  where allow-all-fun f = allow-pfun (Some o f)

definition deny-all-fun :: ('α ⇒ 'β) ⇒ ('α ⇒ 'β) (D₁)
  where deny-all-fun f ≡ deny-pfun (Some o f)

definition deny-all-id :: 'α ⇒ 'α (D₁)
  where deny-all-id ≡ deny-pfun (id o Some)

definition allow-all-id :: 'α ⇒ 'α (A₁)
  where allow-all-id ≡ allow-pfun (id o Some)

definition allow-all :: ('α ⇒ unit) (A₂)
  where allow-all p = ⌊allow ()⌋

definition deny-all :: ('α ⇒ unit) (D₂)
  where deny-all p = ⌊deny ()⌋

... and resulting properties:

lemma A₁ ⊕ Map.empty = A₁
  by simp

lemma A₂ f ⊕ Map.empty = A₂ 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)

16
lemma deny-left-cancel : dom pf = UNIV \implies (deny-pfun pf) \oplus x = (deny-pfun pf)
apply (rule ext)+
by (auto simp: deny-pfun-def option.splits)

2.2.3 Domain, Range, and Restrictions

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

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

whereas range is just an abbreviation for image:

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

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

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

However, some properties are specific to policy concepts:

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

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

lemma ran-deny-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 reorder-
ing of policies composed by override (i.e. by the normal left-to-right rule composition
method.

- Map.dom_map_add dom (?n ⊔ ?m) = dom ?n ∪ dom ?m
- Map.inj_on_map_add_dom inj-on (?m' ⊔ ?m) (dom ?m') = inj-on ?m' (dom ?m')
- Map.map_add_comm dom ?m1.0 ∩ dom ?m2.0 = {} ⇒ ?m2.0 ⊔ ?m1.0 = ?m1.0 ⊔ ?m2.0
- Map.map_add_dom_app_simps(1) ?m ∈ dom ?l2.0 ⇒ (?l2.0 ⊔ ?l1.0) ?m = ?l2.0 ?m
- Map.map_add_dom_app_simps(2) ?m ∉ dom ?l1.0 ⇒ (?l2.0 ⊔ ?l1.0) ?m = ?l2.0 ?m
- Map.map_add_dom_app_simps(3) ?m ∉ dom ?l2.0 ⇒ (?l2.0 ⊔ ?l1.0) ?m = ?l1.0 ?m
- Map.map_add_upd_left ?m ∉ dom ?e2.0 ⇒ ?e2.0 ⊔ ?e1.0(?m ↦ ?u1.0) = (?e2.0 ⊔ ?e1.0)(?m ↦ ?u1.0)

The latter rule also applies to allow- and deny-override.
**definition** dom-restrict :: ['α set, 'α→'β] ⇒ 'α→'β (infixr 55)

**where**  
S ◂ p ≡ (λx. if x ∈ S then p x else ⊥)

**lemma** dom-dom-restrict[simp] : dom(S ◂ p) = S ∩ dom p
**apply** (auto simp: dom-restrict-def)

**subgoal for** x y
  **apply** (case-tac x ∈ S)
  **apply** (simp-all)
  **done

**subgoal for** x y
  **apply** (case-tac x ∈ S)
  **apply** (simp-all)
  **done

**done

**lemma** dom-restrict-idem[simp] : (dom p) ◂ 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 ◂ S ◂ p = T ∩ S ◂ 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 :: ['α→'β, 'β decision set] ⇒ 'α→'β (infixr ⊿ 55)

**where**  
p ⊿ S ≡ (λx. if p x ∈ (Some’S) then p x else ⊥)

**definition** ran-restrict2 :: ['α→'β, 'β decision set] ⇒ 'α→'β (infixr ⊿ 2 55)

**where**  
p ⊿2 S ≡ (λx. if (the (p x)) ∈ (S) then p x else ⊥)

**lemma** ran-restrict = ran-restrict2
**apply** (rule ext)+
**apply** (simp add: ran-restrict-def ran-restrict2-def)

**subgoal for** x xa xb
  **apply** (case-tac x xb)
  **apply** simp-all
  **apply** (metis inj-Some inj-image-mem-iff)
  **done

**done

**20**
lemma ran-ran-restrict[simp] : ran(\(p \triangleright S\)) = \(S \cap \text{ran } p\)
    by (auto simp: ran-restrict-def image-def ran-def)

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

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

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

lemma ran-gen-D[simp] : \((\forall Dx. \lfloor P x \rfloor) \triangleright \text{Deny}\) = \((\forall Dx. \lfloor P x \rfloor)\)
    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]
    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)
    apply (case-tac x)
    apply (rename-tac α)
    apply (case-tac α, simp-all)[1]
    apply (rename-tac α)
    apply (case-tac α, simp-all)[1]
    done

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

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

lemma flat-orD-deny[dest]: flat-orD x = deny y ⇒ x = deny(deny y)
\[
\forall x = \text{allow}(\text{deny } y) \\
\forall x = \text{deny}(\text{allow } y)
\]

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

fun \(\text{flat-1} : (\alpha \text{ decision}) \text{ decision} \Rightarrow (\alpha \text{ decision})\)
where \(\text{flat-1}(\text{allow}(\text{allow } y)) = \text{allow } y\)
| \(\text{flat-1}(\text{allow}(\text{deny } y)) = \text{allow } y\) |
| \(\text{flat-1}(\text{deny}(\text{allow } y)) = \text{deny } y\) |
| \(\text{flat-1}(\text{deny}(\text{deny } y)) = \text{deny } y\) |

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

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

fun \(\text{flat-2} : (\alpha \text{ decision}) \text{ decision} \Rightarrow (\alpha \text{ decision})\)
where \(\text{flat-2}(\text{allow}(\text{allow } y)) = \text{allow } y\)
| \(\text{flat-2}(\text{allow}(\text{deny } y)) = \text{deny } y\) |
| \(\text{flat-2}(\text{deny}(\text{allow } y)) = \text{allow } y\) |
| \(\text{flat-2}(\text{deny}(\text{deny } y)) = \text{deny } y\) |

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

lemma flat-2-deny[dest]: flat-2 x = deny y \Rightarrow x = deny(deny y) \lor 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 -> γ 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 o_m p1)

notation
  comp-orA (infixl oVA 55)

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

lemma mt-comp-orA [simp]: ∅ ◦ A p = ∅
  by (simp add: comp-orA-def)

definition
  comp-orD :: ['β⇒'γ, 'α⇒'β] ⇒ 'α⇒'γ (infixl o'-orD 55) where
  p2 o-orD p1 ≡ (map-option flat-orD) o (lift p2 ◦m p1)

notation
  comp-orD (infixl ◦oRD 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 :: ['β⇒'γ, 'α⇒'β] ⇒ 'α⇒'γ (infixl o'-1 55) where
  p2 o-1 p1 ≡ (map-option flat-1) o (lift p2 ◦m p1)

notation
  comp-1 (infixl ◦1 55)

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

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

definition
  comp-2 :: ['β⇒'γ, 'α⇒'β] ⇒ 'α⇒'γ (infixl o'-2 55) where
  p2 o-2 p1 ≡ (map-option flat-2) o (lift p2 ◦m p1)

notation
  comp-2 (infixl ◦2 55)

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

lemma mt-comp-2 [simp]: ∅ ◦2 p = ∅
  by (simp add: comp-2-def)
2.4 Parallel Composition

theory
  ParallelComposition
imports
  ElementaryPolicies
begin

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

The new input pair will often contain repetitions, which can be reduced using the domain-restriction and domain-reduction operators. Using additional range-modifying operators such as $\nabla$, 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 prod-orA :: $\langle'\alpha \mapsto' \beta, '\gamma \mapsto' \delta\rangle \Rightarrow (\langle '\alpha \times ' \gamma \mapsto ' \beta \times ' \delta \rangle$ (infixr $\otimes_{\otimes_A} 55$)
where

\[ p_1 \otimes_{\otimes_A} p_2 = \]
(\lambda(x,y). \ (case p1 x of 
  [allow d1] \Rightarrow (case p2 y of 
    [allow d2] \Rightarrow [allow(d1,d2)]
    | [deny d2] \Rightarrow [allow(d1,d2)]
    | \bot \Rightarrow \bot)
  | [deny d1] \Rightarrow (case p2 y of 
    [allow d2] \Rightarrow [allow(d1,d2)]
    | [deny d2] \Rightarrow [deny(d1,d2)]
    | \bot \Rightarrow \bot))

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

lemma prod-orA-quasi-commute: p2 ∩ ∨ A p1 = ((λ((x,y). (y,x)) o-f (p1 ∩ ∨ A p2))) o (λ(a,b).(b,a)) 
apply (rule ext) 
apply (simp add: prod-orA-def policy-range-comp-def o-def) 
apply (auto)[1] 
apply (simp split: option.splits decision.splits) 
done

definition prod-orD :: ′α × ′γ ‡ ′δ ⇒ (′α × ′γ ‡ ′δ) (infixr ∩ ∨ D 55)
where p1 ∩ ∨ D p2 = 
(λ(x,y). (case p1 x of 
[allow d1] ⇒ (case p2 y of 
[allow d2] ⇒ [allow(d1,d2)] 
| [deny d2] ⇒ [deny(d1,d2)] 
| ⊥ ⇒ ⊥) 
| [deny d1] ⇒ (case p2 y of 
[allow d2] ⇒ [deny(d1,d2)] 
| [deny d2] ⇒ [deny(d1,d2)] 
| ⊥ ⇒ ⊥) 
| ⊥ ⇒ ⊥))

lemma prod-orD-mt[simp]: p ∩ ∨ D ∅ = ∅ 
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]: ∅ ∩ ∨ D p = ∅ 
apply (rule ext) 
apply (simp add: prod-orD-def) 
done

lemma prod-orD-quasi-commute: p2 ∩ ∨ D p1 = ((λ((x,y). (y,x)) o-f (p1 ∩ ∨ D p2))) o (λ(a,b).(b,a)) 
apply (rule ext) 
apply (simp add: prod-orD-def policy-range-comp-def o-def)
apply (auto)[1]
apply (simp split: option.splits decision.splits)
done

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

**definition** prod-1 :: ['α→'β, 'γ →'δ] ⇒ ('α×'γ → 'β×'δ) (infixr ⊗₁ 55)
where p1 ⊗₁ p2 ≡
(λ(x,y). (case p1 x of
   [allow d1]⇒(case p2 y of
      [allow d2] ⇒ [allow(d1,d2)]
      | [deny d2] ⇒ [allow(d1,d2)]
      | ⊥ ⇒ ⊥)
   | [deny d1] ⇒(case p2 y of
      [allow d2] ⇒ [deny(d1,d2)]
      | [deny d2] ⇒ [deny(d1,d2)]
      | ⊥ ⇒ ⊥)
   | ⊥ ⇒ ⊥))

**lemma** prod-1-mt[simp]: p ⊗₁ ∅ = ∅
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]: ∅ ⊗₁ p = ∅
apply (rule ext)
apply (simp add: prod-1-def)
done

**definition** prod-2 :: ['α→'β, 'γ →'δ] ⇒ ('α×'γ → 'β×'δ) (infixr ⊗₂ 55)
where p1 ⊗₂ p2 ≡
(λ(x,y). (case p1 x of
   [allow d1]⇒(case p2 y of
      [allow d2] ⇒ [allow(d1,d2)]
      | [deny d2] ⇒ [deny(d1,d2)]
      | ⊥ ⇒ ⊥)
   | [deny d1] ⇒(case p2 y of
      [allow d2] ⇒ [deny(d1,d2)]
      | [deny d2] ⇒ [deny(d1,d2)]
      | ⊥ ⇒ ⊥)
   | ⊥ ⇒ ⊥))

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

definition prod-1-id :: (′α ↦→ ′β, ′α ↦→ ′γ) ⇒ (′α ↦→ ′β × ′γ) (infixr \otimes_1 55)
where p \otimes_1 q = (p \otimes_1 q) o (λx. (x,x))

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

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

definition prod-2-id :: (′α ↦→ ′β, ′α ↦→ ′γ) ⇒ (′α ↦→ ′β × ′γ) (infixr \otimes_2 55)
where p \otimes_2 q = (p \otimes_2 q) o (λx. (x,x))

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

lemma mt-prod-2-id[simp]:\∅ \otimes_2 p = \∅
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 :: (′α ⇒ ′β) ⇒ (′δ ⇒ ′γ) ⇒ (′α × ′δ ⇒ ′β × ′γ) (infixr \otimes_\mathcal{M} 60)
where \( p1 \otimes_M p2 = (\lambda (x,y). \text{case } p1 x \text{ of } [d1] \Rightarrow (\text{case } p2 y \text{ of } [d2] \Rightarrow [(d1,d2)] | \bot \Rightarrow \bot)) \)

definition parallel-st :: \((i \times '\sigma \rightarrow '\sigma) \Rightarrow (i \times '\sigma' \rightarrow '\sigma') \Rightarrow (i \times '\sigma \times '\sigma' \rightarrow '\sigma \times '\sigma') \) (infixr \( \otimes \) 60)

where

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

2.4.3 Range Splitting

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

definition comp-ran-split :: \([(\alpha \rightarrow '\gamma) \times (\alpha \rightarrow '\gamma), 'd \rightarrow '\beta] \Rightarrow (\alpha \times '\gamma) \Rightarrow (\beta \times '\gamma) \)

where \( P \otimes_A p \equiv \lambda x. \text{case } p (\text{fst } x) \text{ of } [\text{allow } y] \Rightarrow (\text{case } ((\text{fst } P) (\text{snd } x)) \text{ of } \bot \Rightarrow \bot | [z] \Rightarrow [\text{allow } (y,z)] | [\text{deny } y] \Rightarrow (\text{case } ((\text{snd } P) (\text{snd } x)) \text{ of } \bot \Rightarrow \bot | [z] \Rightarrow [\text{deny } (y,z)] | \bot \Rightarrow \bot) \)

An alternative characterisation of the operator is as follows:

lemma comp-ran-split-charn:

\( (f, g) \otimes_A p = (\text{prod } 1 A_p) \oplus \text{map } add \text{ add } o \text{ def } \text{ran } \text{restrict } \text{def } \text{image } \text{def } \text{allow } \text{def } \text{Deny } \text{def } \text{dom } \text{restrict } \text{def } \text{prod-or } \text{def } \text{allow-pfun } \text{def } \text{deny-pfun } \text{def } \text{split } \text{option } \text{splits } \text{decision } \text{splits} \) apply (auto)
done

2.4.4 Distributivity of the parallel combinators

lemma distr-or1-a: \( (F = F1 \oplus F2) \Rightarrow (((N \otimes_1 F) \circ f) = ((N \otimes_1 F1) \circ f) \oplus ((N \otimes_1 F2) \circ f))) \)

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

lemma distr-or2: ($F = F1 \oplus F2$)\implies ((r \circ f \ ((N \otimes_2 F) \circ f)) =
((r \circ f \ ((N \otimes_2 F1) \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_{VA} F) \circ f)) =
\begin{verbatim}
apply (rule ext)+
apply (simp add: prod-orA-def map-add-def policy-range-comp-def
    split: decision.splits option.splits)
subgoal for x
  apply (case-tac f x)
  apply (simp-all add: map-add-def
    split: decision.splits option.splits)
done
done

lemma distr-orD: \( (F = F1 \oplus F2) \implies ((g \circ f ((N \times_{A} F1) \circ f)) = ((g \circ f ((N \times_{D} F1) \circ f)) \oplus (g \circ f ((N \times_{D} F2) \circ f))) \)
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

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.

\end{verbatim}
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 ⊑ A 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 cxE)
  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 ⊑ D q = (∀ x. (case q x of ⌊deny y⌋ ⇒ (∃ z. (p x = ⌊deny z⌋)))
                            | [allow y] ⇒ True
                            | ⊥ ⇒ True)

lemma more-rejective-trans : p ⊑ D p' ⇒ p' ⊑ D p'' ⇒ p ⊑ D 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 ⊑ D p
unfolding more-rejective-def
by(auto split : option.split decision.split)

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

lemma A I ⊑ 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 ⊑ absa,absb q ≡
  (∀ a. case p a of ⊥ ⇒ True
                 | [allow y] ⇒ (∀ a'∈{x. absa x=a}. 
                         ∃ b'. q a' = [allow b']
                         ∧ absb b' = y)
                 | [deny y] ⇒ (∀ a'∈{x. absa x=a}. 

34
∃ b'. q a' = [deny b']
∧ abs b' = y))

**Theorem** polref-refl: p ⊑ \text{id} p

**Unfolding** policy-refinement-def

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

**Theorem** polref-trans:

- assumes A: p ⊑ f.g p'
  - and B: p' ⊑ f'.g' p''
- shows p ⊑ f o f o 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 :: \( (\forall a \to b) \Rightarrow (\forall a \to b) \Rightarrow \text{bool} \) where

\[ p \approx D q = (\forall x \in D. p x = q x) \]

p and q have no conflicts

**Definition** no-conflicts :: \( (\forall a \to b) \Rightarrow (\forall a \to b) \Rightarrow \text{bool} \) where

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

**Lemma** policy-eq:

- assumes p-over-qA: p ⊑ A q
- and q-over-pA: q ⊑ A p
- and p-over-qD: q ⊑ D p
- and q-over-pD: p ⊑ D q
- and dom-eq: dom p = dom q

35
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)
done

Miscellaneous

lemma dom-inter: [[dom p ∩ dom q = {}]: 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 poli-
cies. A typical usage scenario can be observed in the firewall case study [12].

36
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

\[
\text{list2policy } l = \text{foldr } (\lambda x y. (x \oplus y)) \ l \ \emptyset
\]

Determine the position of element of a list.

**fun position :: 'a ⇒ 'a list ⇒ nat where**

\[
\text{position } a \ [] = 0
\]

\[
\text{(position } a \ (x#xs)) = (\text{if } a = x \text{ then } 1 \text{ else } (\text{Suc } (\text{position } a \ xs)))
\]

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

**fun applied-rule where**

\[
\text{applied-rule } C \ a \ (x#xs) = (\text{if } a \in \text{dom } (C \ x) \text{ then } \text{Some } x \text{ else } \text{applied-rule } C \ a \ xs)
\]

|applied-rule C a [] = None

The following is used if the list is constructed backwards.

**definition applied-rule-rev where**

\[
\text{applied-rule-rev } C \ a \ x = \text{applied-rule } C \ a \ (\text{rev } x)
\]

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**

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

|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**

\[
\text{none-MT-rules } C \ (x#xs) = (\text{dom } (C \ x) \neq \{\} \wedge \text{none-MT-rules } C \ xs)
\]

|none-MT-rules C [] = True

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

**fun not-MT where**

\[
\text{not-MT } C \ (x#xs) = (\text{if } \text{dom } (C \ x) = \{\} \text{ then } \text{not-MT } C \ xs \text{ else True})
\]
Next, a few theorems about the two invariants and the transformation:

**Lemma none-MT-rules-vs-notMT**: none-MT-rules C p \(\implies\) p \(\neq\) [] \(\implies\) 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 \(\implies\) (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 \(\implies\) 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] \(\implies\) dom (C a) \(\neq\) {}
apply (simp add: nMTcharn)
done

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

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

**Lemma not-MTimpnotMT**: not-MT C p \(\implies\) p \(\neq\) []
apply (auto)
done
lemma SR3nMT: \( \neg \text{not-MT } C \ p \Rightarrow \text{rm-MT-rules } C \ p = [] \)
apply (induct p)
apply (auto simp: if-splits)
done

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

lemma NMPrm: \( \text{not-MT } C \ p \Rightarrow \text{not-MT } C \ (\text{rm-MT-rules } C \ p) \)
apply (induct p)
apply (simp-all)
done

Next, a few theorems about applied_rule:

lemma mrconc: \( \text{applied-rule-rev } C \ x \ p = \text{Some } a \Rightarrow \text{applied-rule-rev } C \ x \ (\#b p) = \text{Some } a \)
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: \([ \text{applied-rule-rev } C \ x \ b = \text{Some } r ; \text{applied-rule-rev } C \ x \ c = \text{Some } r ] \Rightarrow \text{applied-rule-rev } C \ x \ (a\#b) = \text{applied-rule-rev } C \ x \ (a\#c) \)
by (simp add: mrconc)

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

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

\subsection*{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.

\[
\text{fun } \text{prod-list} :: (\alpha \mapsto \beta) \Rightarrow ((\gamma \mapsto \delta) \text{ list}) \Rightarrow \\
((\alpha \mapsto \beta) \Rightarrow (\gamma \mapsto \delta) \Rightarrow ((\alpha \times \gamma) \mapsto ((\beta \times \delta))) \Rightarrow \\
((\beta \times \delta) \Rightarrow y) \Rightarrow ((\alpha \times \gamma)) \Rightarrow \\
((\alpha \times \gamma) \mapsto y) \text{ list}) \quad \text{infixr } \times \\n\]


where

\[
\text{prod-list} x \ (y \# y) \parallel \text{comb ran-adapt dom-adapt} = \\
((\text{ran-adapt } o \text{-f } ((\parallel \text{comb } x \ y) \ o \text{ dom-adapt}))\#(\text{prod-list } x \ y) \parallel \text{comb ran-adapt dom-adapt}) \\
\]

An instance, as usual there are four of them.

\[
\text{definition} \ \text{prod-2-list} :: [(\alpha \mapsto \beta), ((\gamma \mapsto \delta) \text{ list})] \Rightarrow \\
((\beta \times \delta) \Rightarrow y) \Rightarrow ((\alpha \times \gamma)) \Rightarrow \\
((\alpha \times \gamma) \mapsto y) \text{ list}) \quad \text{infixr } \otimes \\n\]

\[
x \otimes \ y = (\lambda d \ r. \ (x \otimes \ y) (\otimes \ 2 \ d \ r) \\
\]

\[
\text{lemma} \ \text{list2listNMT}: x \neq [] \implies \text{map sem } x \neq [] \\
\text{apply (case-tac } x) \\
\text{apply (simp-all)} \\
\text{done} \\
\]

\[
\text{lemma} \ \text{two-conc}: (\text{prod-list } x \ (y \# y) \ p \ r \ d) = ((r \ o \text{-f } ((p \ x \ y) \ o \text{ dom-adapt}))\#(\text{prod-list } x \ y) \parallel \text{comb ran-adapt dom-adapt}) \\
\text{by simp} \\
\]

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

\[
\text{definition} \ \text{is-distr where} \\
is-distr p = (\lambda \ g \ f. \ (\forall N \ P1 \ P2. ((g \ o \text{-f } ((p \ N \ P1 ) \uplus P2)) \ o \ f)) = \\
((g \ o \text{-f } ((p \ N \ P1 ) \ o \ f)) \uplus (g \ o \text{-f } ((p \ N \ P2 ) \ o \ f)))) \\
\]

\[
\text{definition} \ \text{is-strict where} \\
is-strict p = (\lambda r \ d. \ (r \ o \text{-f } (p \ P1 ) \ o \text{ dom-adapt}) \ o d) = \emptyset \\
\]

\[
\text{lemma} \ \text{is-distr-orD}: \text{is-distr } (\otimes \lor D) \ d \ r \\
\text{apply (simp add: is-distr-def)} \\
\text{apply (rule allI)+} \\
\text{apply (rule distr-orD)} \\
\text{apply (simp)} \\
\]
The following theorems are crucial: they establish the correctness of the distribution.

```
lemma is-strict-orD: is-strict (⨂ ∨ D) d r
  apply (simp add: is-strict-def)
  apply (simp add: policy-range-comp-def)
  done

lemma is-distr-2: is-distr (⨂ 2) d r
  apply (simp add: is-distr-def)
  apply (rule allI)+
  apply (rule distr-or2)
  by simp

lemma is-strict-2: is-strict (⨂ 2) d r
  apply (simp only: is-strict-def)
  apply simp
  apply (simp add: policy-range-comp-def)
  done

lemma domStart: t ∈ dom p1 ⇒ (p1 ⨁ p2) t = p1 t
  apply (simp add: map-add-dom-app-simps)
  done

lemma notDom: x ∈ dom A ⇒ ¬ A x = None
  apply auto
  done

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

```

lemma Norm-Distr-1: ((r o-f (((⨂ 1) P1 (list2policy P2)) o d)) x = ((list2policy ((P1 ⊗ L P2) (⨂ 1) r d)) x))
proof (induct P2)
  case Nil show ?case
    by (simp add: policy-range-comp-def list2policy-def)
next
  case (Cons p ps) show ?case using Cons
  proof (cases x ∈ dom (r o-f ((P1 ⊗ 1 p) o d)))
    case True show ?thesis using True
      by (auto simp: list2policy-def policy-range-comp-def prod-1-def
            split: option.splits decision.splits prod.splits)
  next
  case False show ?thesis using Cons False
    by (auto simp: list2policy-def policy-range-comp-def map-add-dom-app-simps(3)
            prod-1-def
            split: option.splits decision.splits prod.splits)
```
lemma Norm-Distr-2: \((r \circ f \ (((\times_2 P1) \ (list2policy P2)) \ o d))\) \(x = \ ((list2policy ((P1 \times L P2) \ (\times_2 r d))) \ x)\)
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 \(x \in \text{dom} \ (r \circ f \ ((P1 \times_2 p) \ o d))\))
    case True show ?thesis using True
    by (auto simp: list2policy-def prod-2-def policy-range-comp-def
        split: option.splits decision.splits prod.splits)
  next
    case False show ?thesis using Cons False
    by (auto simp: policy-range-comp-def list2policy-def map-add-dom-app-simps(3)
        prod-2-def
        split: option.splits decision.splits prod.splits)
  qed
qed

lemma Norm-Distr-A: \((r \circ f \ (((\mathbin{\lor_A} P1) \ (list2policy P2)) \ o d))\) \(x = \ ((list2policy ((P1 \times L P2) \ (\mathbin{\lor_A} r d))) \ x)\)
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 \(x \in \text{dom} \ (r \circ f \ ((P1 \times \mathbin{\lor_A} p) \ o d))\))
    case True show ?thesis using True
    by (auto simp: policy-range-comp-def list2policy-def prod-orA-def
        split: option.splits decision.splits prod.splits)
  next
    case False show ?thesis using Cons False
    by (auto simp: policy-range-comp-def list2policy-def map-add-dom-app-simps(3)
        prod-orA-def
        split: option.splits decision.splits prod.splits)
  qed
qed

lemma Norm-Distr-D: \((r \circ f \ (((\mathbin{\lor_D} P1) \ (list2policy P2)) \ o d))\) \(x = \ ((list2policy ((P1 \times L P2) \ (\mathbin{\lor_D} r d))) \ x)\)
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 ⨇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.split 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.split splits)
qed

Some domain reasoning

lemma domSubsetDistr1: dom A = UNIV =⇒ dom ((λ x. 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.split option.split splits)
done

lemma domSubsetDistr2: dom A = UNIV =⇒ dom ((λ x. 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.split option.split splits)
done

lemma domSubsetDistrA: dom A = UNIV =⇒ dom ((λ x. 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.split option.split splits)
done

lemma domSubsetDistrD: dom A = UNIV =⇒ dom ((λ x. 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 ∈ dom P ⇒ (list2policy PL) x = P x ⇒ disjDom PL ⇒ (PUT x = P x) ⇒
PUTList PUT x PL
apply (induct PL)
apply (simp-all add: list2policy-def)
apply (auto)[1]
subgoal for a PL p
  apply (case-tac x ∈ 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 ∈ dom P; (list2policy PL) x = P x; disjDom PL ⇒ (PUT x = P x) ⇒
PUTList PUT x PL
apply (rule iffI)
apply (rule distrPUTL2)
  apply (simp-all)
apply (rule-tac PL = PL in distrPUTL1)
  apply (auto)
done

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

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

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

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

lemma distrPUTLG1: [x ∈ dom P; (list2policy PL) x = P x; PUTListGD PUT x PL] \implies PUT x = P x
  apply (induct PL)
  apply (simp-all add: domIff PUTListGD-def disjDomGD-def gatherDomain-def list2policy-def)
  apply (auto simp: dom-def domIff split: if-split-asm)
done

lemma distrPUTLG2:
  PL ≠ [] \implies x ∈ dom P \implies (list2policy (PL)) x = P x \implies disjDomGD PL \implies (PUT x = P x) \implies PUTListGD PUT x (PL)
  apply (simp add: PUTListGD-def disjDomGD-def gatherDomain-def list2policy-def)
  apply (induct PL)
  apply (auto)
  apply (metis PUTList-DomMT PUTList-None domI)
done

lemma distrPUTLG:
  [x ∈ dom P; (list2policy PL) x = P x; disjDomGD PL; PL ≠ []] \implies (PUT x = P x) = PUTListGD PUT x PL
  apply (rule iffI)
  apply (rule distrPUTLG2)
  apply (simp-all)
  apply (rule-tac PL = PL in distrPUTLG1)
  apply (auto)
done

end

2.8 Putting Everything Together: UPF

theory UPF
  imports
    Normalisation
    NormalisationTestSpecification
    Analysis
begin

This is the top-level theory for the Unified Policy Framework (UPF) and, thus, builds
the base theory for using UPF. For the moment, we only define a set of lemmas for all
core UPF definitions that is useful for using UPF:
lemmas $UPFDefs = UPFCoreDefs\ ParallelDefs\ ElementaryPoliciesDefs$
end
3 Example

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

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

3.1 Secure Service Specification

theory
  Service
imports
  UPF
begin

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

3.1.1 Datatypes for Modelling Users and Roles

Users

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

type-synonym user = int

Similarly, each patient has a unique id:

type-synonym patient = int

Roles and Relationships

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

datatype role = ClinicalPractitioner | Nurse | Clerical

We model treatment relationships (legitimate relationships) between staff and patients (respectively, their health records. This access control model is inspired by our detailed NHS model.
type-synonym \( lr-id = \text{int} \)

**The security context stores all the existing LRs.**

**type-synonym \( \Sigma = \text{patient} \rightarrow \text{LR} \)**

The user context stores the roles the users are in.

**type-synonym \( \nu = \text{user} \rightarrow \text{role} \)**

### 3.1.2 Modelling Health Records and the Web Service API

**Health Records**

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

**datatype \( \text{data} \) = \text{dummyContent}**

**datatype \( \text{status} \) = \text{Open} | \text{Closed}**

**type-synonym \( \text{entry-id} = \text{int} \)**

**type-synonym \( \text{entry} = \text{status} \times \text{user} \times \text{data} \)**

**type-synonym \( \text{SCR} = (\text{entry-id} \rightarrow \text{entry}) \)**

**type-synonym \( \text{DB} = \text{patient} \rightarrow \text{SCR} \)**

**The Web Service API**

The operations provided by the service:

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

**fun \( \text{is-createSCR where} \)**

\( \text{is-createSCR (createSCR u r p)} = \text{True} \)
\( | \text{is-createSCR} x = \text{False} \)

**fun \( \text{is-appendEntry where} \)**

\( \text{is-appendEntry (appendEntry u r e ei)} = \text{True} \)
\( | \text{is-appendEntry} x = \text{False} \)

**fun \( \text{is-deleteEntry where} \)**
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 patientOfOp :: Operation ⇒ user where
patientOfOp (appendEntry u r p e ei) = p
patientOfOp (deleteEntry u r p e-id) = p
patientOfOp (readEntry u r p e) = p
patientOfOp (readSCR u r p) = p
patientOfOp (changeStatus u r p s ei) = p
patientOfOp (deleteSCR u r p) = p
patientOfOp (addLR u r p lr ei) = p
patientOfOp (removeLR u r p lr) = p
patientOfOp (editEntry u r p e-id s) = p

fun userOfOp :: Operation ⇒ role 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) = (snd (snd e))
contentOfOp (editEntry u r p e ei) = (snd (snd e))

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

fun allContentStatic where
3.1.3 Modelling Access Control

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

Sealed Envelopes

type-synonym SEPolicy = (Operation \times DB \mapsto \text{unit})

definition get-entry:: DB \Rightarrow \text{patient} \Rightarrow \text{entry-id} \Rightarrow \text{entry option} where
get-entry S p e-id = (case S p of \bot \Rightarrow \bot
\mid [\text{Scr}] \Rightarrow \text{Scr e-id})

definition userHasAccess:: user \Rightarrow \text{entry} \Rightarrow \text{bool} where
userHasAccess u e = ((\text{fst } e) = \text{Open} \lor (\text{fst } (\text{snd } e) = u))

definition readEntrySE :: SEPolicy where
readEntrySE x = (case x of (readEntry u r p e-id,S) \Rightarrow (\text{case get-entry } S p e-id of
\bot \Rightarrow \bot
\mid [e] \Rightarrow (\text{if } (\text{userHasAccess } u e)
\text{then } (\text{allow } ())
\text{else } (\text{deny } ()))))
\mid x \Rightarrow \bot)

definition deleteEntrySE :: SEPolicy where
deleteEntrySE x = (case x of (deleteEntry u r p e-id,S) \Rightarrow (\text{case get-entry } S p e-id of
\bot \Rightarrow \bot
\mid [e] \Rightarrow (\text{if } (\text{userHasAccess } u e)
\text{then } (\text{allow } ())
\text{else } (\text{deny } ()))))
\mid x \Rightarrow \bot)

definition editEntrySE :: SEPolicy where
editEntrySE x = (case x of (editEntry u r p e-id s,S) \Rightarrow (\text{case get-entry } S p e-id of
\bot \Rightarrow \bot
\mid [e] \Rightarrow (\text{if } (\text{userHasAccess } u e)
\text{then } (\text{allow } ())
\text{else } (\text{deny } ()))))
definition $SEPolicy :: SEPolicy$ where
$SEPolicy \equiv editEntrySE \oplus deleteEntrySE \oplus readEntrySE \oplus A_U$

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

Legitimate Relationships

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

fun hasLR :: user $\Rightarrow$ patient $\Rightarrow$ $\Sigma$ $\Rightarrow$ bool where
$hasLR\ u\ p\ \Sigma = (\text{case } \Sigma\ p\ \text{of } \bot\ \Rightarrow False$
$| lrs\ \Rightarrow (\exists \ lr.\ lr\in(\text{ran} \ lrs) \land u\in lr))$

definition $LRPolicy :: LRPolicy$ where
$LRPolicy = (\lambda (x,y).\ (\text{if } hasLR (\text{userOfOp} \ x)\ (\text{patientOfOp} \ x)\ y$
$\text{ then } [allow ()]\$
$\text{ else } [deny ()]))$

definition $createSCRPolicy :: LRPolicy$ where
$createSCRPolicy\ x = (\text{if } (\text{is-createSCR} \ (\text{fst} \ x))$
$\text{ then } [allow ()]\$
$\text{ else } \bot)$

definition $addLRPolicy :: LRPolicy$ where
$addLRPolicy\ x = (\text{if } (\text{is-addLR} \ (\text{fst} \ x))$
$\text{ then } [allow ()]\$
$\text{ else } \bot)$

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

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

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

fun createFunPolicy :: FunPolicy where
$createFunPolicy\ ((createSCR\ u\ r\ p),(D,S)) = (\text{if } p\in \text{dom} \ D$
$\text{ then } [deny ()]\$
$\text{ else } [allow ()])$
$| createFunPolicy\ x = \bot$
fun addLRFunPolicy :: FunPolicy where
addLRFunPolicy ((addLR u r p l us),(D,S)) = (if l \in \text{dom } S
then \lfloor \text{deny }() \rfloor
else \lfloor \text{allow }() \rfloor)
|addLRFunPolicy x = \bot

fun removeLRFunPolicy :: FunPolicy where
removeLRFunPolicy ((removeLR u r p l),(D,S)) = (if l \in \text{dom } S
then \lfloor \text{allow }() \rfloor
else \lfloor \text{deny }() \rfloor)
|removeLRFunPolicy x = \bot

fun readSCRFunPolicy :: FunPolicy where
readSCRFunPolicy ((readSCR u r p),(D,S)) = (if p \in \text{dom } D
then \lfloor \text{allow }() \rfloor
else \lfloor \text{deny }() \rfloor)
|readSCRFunPolicy x = \bot

fun deleteSCRFunPolicy :: FunPolicy where
deleteSCRFunPolicy ((deleteSCR u r p),(D,S)) = (if p \in \text{dom } D
then \lfloor \text{allow }() \rfloor
else \lfloor \text{deny }() \rfloor)
|deleteSCRFunPolicy x = \bot

fun changeStatusFunPolicy :: FunPolicy where
changeStatusFunPolicy (changeStatus u r p e s,(d,S)) =
\begin{align*}
\text{case } d p \text{ of } \lfloor x \rfloor \Rightarrow
\begin{cases}
\text{if } e \in \text{dom } x \\
\text{then } \lfloor \text{allow }() \rfloor \\
\text{else } \lfloor \text{deny }() \rfloor
\end{cases}
| - \Rightarrow \lfloor \text{deny }() \rfloor
\end{align*}
|changeStatusFunPolicy x = \bot

fun deleteEntryFunPolicy :: FunPolicy where
deleteEntryFunPolicy (deleteEntry u r p e,(d,S)) =
\begin{align*}
\text{case } d p \text{ of } \lfloor x \rfloor \Rightarrow
\begin{cases}
\text{if } e \in \text{dom } x \\
\text{then } \lfloor \text{allow }() \rfloor \\
\text{else } \lfloor \text{deny }() \rfloor
\end{cases}
| - \Rightarrow \lfloor \text{deny }() \rfloor
\end{align*}
|deleteEntryFunPolicy x = \bot

fun readEntryFunPolicy :: FunPolicy where
readEntryFunPolicy (readEntry u r p e,(d,S)) =
\begin{align*}
\text{case } d p \text{ of } \lfloor x \rfloor \Rightarrow
\begin{cases}
\text{if } e \in \text{dom } x
\end{cases}
\end{align*}
then \(\text{allow}()\)
else \(\text{deny}()\)
\(|\text{den}y()\)
|\(\text{readEntryFunPolicy} \ x = \bot\)

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

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

**definition FunPolicy where**
FunPolicy = editEntryFunPolicy \(\oplus\) appendEntryFunPolicy
\(\oplus\) readEntryFunPolicy \(\oplus\) deleteEntryFunPolicy
\(\oplus\) changeStatusFunPolicy \(\oplus\) deleteSCRFunPolicy
\(\oplus\) removeLRFunPolicy \(\oplus\) readSCRFunPolicy
\(\oplus\) addLRFunPolicy \(\oplus\) createFunPolicy \(\oplus\) \(\bot\)

**Modelling Core RBAC**

**type-synonym RBACPolicy = Operation \times \nu \mapsto \text{unit}**

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

definition RBACPolicy :: RBACPolicy where
RBACPolicy = (\lambda (f, uc). if \ ((\text{roleOfOp} \ f) \in \text{RBAC} \land [\text{roleOfOp} \ f] = uc \ (\text{userOfOp} \ f)) then \ [\text{allow} ()] else \ [\text{deny} ()])

3.1.4 The State Transitions and Output Function

State Transition

fun OpSuccessDB :: (Operation \times DB) \rightarrow DB \ where
OpSuccessDB (createSCR u r p S) = (case S p of \perp \Rightarrow [S(p \mapsto \emptyset)]
| [x] \Rightarrow [S])
|OpSuccessDB ((appendEntry u r p ei e),S) =
\begin{align*}
\text{(case S p of } \perp \Rightarrow [S] \\
| [x] \Rightarrow \text{(if } ei \in (\text{dom} \ x) \text{ then } [S] \text{ else } [S(p \mapsto x(ei\mapsto e))])\text{)}
\end{align*}
|OpSuccessDB ((deleteSCR u r p),S) = (\text{Some } (S(p \mapsto \perp)))
|OpSuccessDB ((deleteEntry u r p ei),S) =
\begin{align*}
\text{(case S p of } \perp \Rightarrow [S] \\
| [x] \Rightarrow \text{Some } (S(p \mapsto x(ei\mapsto e))))
\end{align*}
|OpSuccessDB ((changeStatus u r p ei s),S) =
\begin{align*}
\text{(case S p of } \perp \Rightarrow [S] \\
| [x] \Rightarrow (\text{case } x \ ei \ of} \\
| e \Rightarrow [S(p \mapsto x(ei\mapsto (s, \text{snd} \ e)))] \\
| \perp \Rightarrow [S])
\end{align*}
|OpSuccessDB ((editEntry u r p ei e),S) =
\begin{align*}
\text{(case S p of } \perp \Rightarrow [S] \\
| [x] \Rightarrow (\text{case } x \ ei \ of} \\
| e \Rightarrow [S(p \mapsto x(ei\mapsto (e)))] \\
| \perp \Rightarrow [S])
\end{align*}
|OpSuccessDB (x,S) = [S]

fun OpSuccessSigma :: (Operation \times \Sigma) \rightarrow \Sigma \ where
OpSuccessSigma (addLR u r p lr-id us,S) =
\begin{align*}
\text{(case S p of } [\text{lrs}] \Rightarrow (\text{case } (\text{lrs lr-id}) \ of} \\
| \perp \Rightarrow [S(p \mapsto (\text{lrs lr-id} \mapsto us))] \\
| [x] \Rightarrow [S])
\end{align*}
|OpSuccessSigma (removeLR u r p lr-id,S) =
(case \(S \ p\) of Some \(lrs\) \(\Rightarrow \lfloor S(p \mapsto (lrs(lr-id:=\bot))\rfloor\))

| \(\bot \Rightarrow \lfloor S\rfloor\) |

| \(\text{OpSuccessSigma} \ (x,S) = \lfloor S\rfloor\) |

fun \(\text{OpSuccessUC} \ :: \ (\text{Operation} \times \nu) \rightarrow \nu\) where

\(\text{OpSuccessUC} \ (f,u) = \lfloor u\rfloor\)

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

fun \(\text{OpSuccessOutput} \ :: \ (\text{Operation}) \rightarrow \text{Output}\) where

\(\text{OpSuccessOutput} \ x = \lfloor ()\rfloor\)

fun \(\text{OpFailOutput} \ :: \ \text{Operation} \rightarrow \text{Output}\) where

\(\text{OpFailOutput} \ x = \lfloor ()\rfloor\)

3.1.5 Combine All Parts
definition \(\text{SE-LR-Policy} \ :: \ (\text{Operation} \times \text{DB} \times \Sigma, \text{unit})\) policy where

\(\text{SE-LR-Policy} = (\lambda (x,x). \ x) \ o_f (\text{SEPolicy} \ \odot_D \ \text{LR-Policy}) \ o (\lambda (a,b,c). ((a,b),a,c))\)

definition \(\text{SE-LR-FUN-Policy} \ :: \ (\text{Operation} \times \text{DB} \times \Sigma, \text{unit})\) policy where

\(\text{SE-LR-FUN-Policy} = ((\lambda (x,x). \ x) \ o_f (\text{FunPolicy} \ \odot_D \ \text{SE-LR-Policy}) \ o (\lambda a. (a,a)))\)

definition \(\text{SE-LR-RBAC-Policy} \ :: \ (\text{Operation} \times \text{DB} \times \Sigma \times \nu, \text{unit})\) policy where

\(\text{SE-LR-RBAC-Policy} = (\lambda (x,x). \ x) \ o_f (\text{RBACPolicy} \ \odot_D \ \text{SE-LR-FUN-Policy}) \ o (\lambda (a,b,c,d). ((a,d),(a,b,c)))\)

definition \(\text{ST-Allow} \ :: \ \text{Operation} \times \text{DB} \times \Sigma \times \nu \rightarrow \text{Output} \times \text{DB} \times \Sigma \times \nu\) where

\(\text{ST-Allow} = ((\text{OpSuccessOutput} \ M (\text{OpSuccessDB} \ S \ \text{OpSuccessSigma} \ S \ \text{OpSuccessUC}) \ o (\lambda (a,b,c). ((a),(a,b,c))))\)

definition \(\text{ST-Deny} \ :: \ \text{Operation} \times \text{DB} \times \Sigma \times \nu \rightarrow \text{Output} \times \text{DB} \times \Sigma \times \nu\) where

\(\text{ST-Deny} = (\lambda (\text{ope,sp,si,uc}). \text{Some} (((), \text{sp,si,uc})))\)

definition \(\text{SE-LR-RBAC-ST-Policy} \ :: \ \text{Operation} \times \text{DB} \times \Sigma \times \nu \mapsto \text{Output} \times \text{DB} \times \text{DB} \times \Sigma \times \nu\)
\[ \Sigma \times \upsilon \]
where

\[
SE-LR-RBAC-ST-Policy = ( (\lambda (x,y).y) \\
of ((ST-Allow,ST-Deny) \otimes \nabla SE-LR-RBAC-Policy) \\
o (\lambda x.(x,x)))
\]

**definition**

\[ PolMon :: Operation \Rightarrow (Output \ \text{decision}, DB \times \Sigma \times \upsilon) \ \text{MON}_SE \]

where

\[ PolMon = (policy2MON \ SE-LR-RBAC-ST-Policy) \]

**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**

\[ alice :: user \ \text{where} \ \ alice = 1 \]

**definition**

\[ bob :: user \ \text{where} \ \ bob = 2 \]

**definition**

\[ charlie :: user \ \text{where} \ \ charlie = 3 \]

**definition**

\[ patient1 :: patient \ \text{where} \ \ patient1 = 5 \]

**definition**

\[ patient2 :: patient \ \text{where} \ \ patient2 = 6 \]

**definition**

\[ UC0 :: \upsilon \ \text{where} \]

\[ UC0 = \text{Map}.\emptyset(alice\mapsto\text{Nurse})(bob\mapsto\text{ClinicalPractitioner})(charlie\mapsto\text{Clerical}) \]

**definition**

\[ entry1 :: entry \ \text{where} \]

\[ entry1 = (Open,alice, dummyContent) \]

**definition**

\[ entry2 :: entry \ \text{where} \]

\[ entry2 = (Closed,bob, dummyContent) \]

**definition**

\[ entry3 :: entry \ \text{where} \]

\[ entry3 = (Closed,alice, dummyContent) \]

**definition**

\[ SCR1 :: SCR \ \text{where} \]

\[ SCR1 = (\text{Map}.\emptyset(1\mapsto entry1)) \]
definition \( \text{SCR2} :: \text{SCR} \) where
\( \text{SCR2} = (\text{Map}.\text{empty}) \)

definition \( \text{Spine0} :: \text{DB} \) where
\( \text{Spine0} = \text{Map}.\text{empty}(\text{patient1} \mapsto \text{SCR1})(\text{patient2} \mapsto \text{SCR2}) \)

definition \( \text{LR1} :: \text{LR} \) where
\( \text{LR1} = (\text{Map}.\text{empty}(1 \mapsto \{\text{alice}\})) \)

definition \( \Sigma0 :: \Sigma \) where
\( \Sigma0 = (\text{Map}.\text{empty}(\text{patient1} \mapsto \text{LR1})) \)

3.2.2 The Initial System State

definition \( \sigma0 :: \text{DB} \times \Sigma \times \upsilon \) where
\( \sigma0 = (\text{Spine0}, \Sigma0, \text{UC0}) \)

3.2.3 Basic Properties

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

lemma [cong,simp]:
\(((\text{if hasLR } \text{urp1-alice } 1 \Sigma0 \text{ then } \text{allow } () \text{ else } \text{deny } () = \bot) = \text{False} \)
by \((\text{simp})\)

lemmas \( \text{MonSimps} = \text{valid-SE-def} \text{ unit-SE-def} \text{ bind-SE-def} \)
lemmas \( \text{Psplits} = \text{option.splits} \text{ unit.splits} \text{ prod.splits} \text{ decision.splits} \)
lemmas \( \text{PolSimps} = \text{valid-SE-def} \text{ unit-SE-def} \text{ bind-SE-def} \text{ if-splits} \text{ policy2MON-def} \)
\( \text{SE-LR-RBAC-ST-Policy-def} \text{ map-add-def} \text{ id-def} \text{ LRsimps} \text{ prod-2-def} \)
\( \text{RBACPolicy-def} \)
\( \text{SE-LR-Policy-def} \text{ SEPolicy-def} \text{ RBAC-def} \text{ deleteEntrySE-def} \text{ editEntrySE-def} \)
\( \text{readEntrySE-def} \sigma0-def \Sigma0-def \text{ UC0-def} \text{ patient1-def} \text{ patient2-def} \text{ LR1-def} \)
\( \text{alice-def bob-def charlie-def get-entry-def SE-LR-RBAC-Policy-def} \text{ Allow-def} \)
\( \text{Deny-def} \text{ dom-restrict-def} \text{ policy-range-comp-def} \text{ prod-orA-def} \text{ prod-orD-def} \)
\( \text{ST-Allow-def} \text{ ST-Deny-def} \text{ Spine0-def} \text{ SCR1-def} \text{ SCR2-def} \text{ entry1-def} \)
\( \text{entry2-def} \)
\( \text{entry3-def} \text{ FunPolicy-def} \text{ SE-LR-FUN-Policy-def} \text{ o-def} \text{ image-def} \text{ UPFDefs} \)
lemma \textit{SE-LR-RBAC-Policy} (\((\text{createSCR alice Clerical patient1}),\sigma_0\)) = \textit{Some (deny ()\)}
   by (simp add: PolSimps)

lemma \textit{exBool[simp]}: \(\exists \ a::\text{bool}.\ a\)
   by auto

lemma \textit{deny-allow[simp]}: \([\text{deny ()}] \notin \textit{Some \ ' range allow}\)
   by auto

lemma \textit{allow-deny[simp]}: \([\text{allow ()}] \notin \textit{Some \ ' range deny}\)
   by auto

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

lemma \(\sigma_0 \models (\text{os} \leftarrow \text{mbind} \ ((\text{createSCR alice Clerical patient1})) \ (\text{PolMon});\)
   \(\text{return} \ (\text{os} = \ [(\text{deny (Out)})])\)))
   by (simp add: PolMon-def MonSimps PolSimps)

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

lemma \textit{SE-LR-RBAC-Policy} (\((\text{appendEntry alice Clerical patient1 ei d}),\sigma_0\)) = \([\text{deny ()}]\)
   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 (′o, ′σ) MONSE = ′σ ⇒ (′o × ′σ)
definition bind-SE :: (′o,′σ)MONSE ⇒ (′o ⇒ (′o,′σ)MONSE) ⇒ (′o,′σ)MONSE
where  bind-SE f g = (λσ. case f σ of None ⇒ None
                             | Some (out, σ′) ⇒ g out σ′)
notation bind-SE (bindSE)
syntax
-bind-SE :: [pttrn,'(o','σ)MON_SE,,'(α','σ)MON_SE] ⇒ ('α','σ)MON_SE

translations

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

**definition** unit-SE :: 'o ⇒ ('o,'σ)MON_SE
**where** \[ \text{unit-SE e = (λσ. Some(e,σ))} \]
**notation** unit-SE (unit_SE)

**definition** fail_SE :: ('o,'σ)MON_SE
**where** \[ \text{fail_SE = (λσ. None)} \]
**notation** fail_SE (fail_SE)

**definition** assert-SE :: ('σ ⇒ bool) ⇒ (bool,'σ)MON_SE
**where** \[ \text{assert-SE P = (λσ. if P σ then Some(True,σ) else None)} \]
**notation** assert-SE (assert_SE)

**definition** assume-SE :: ('σ ⇒ bool) ⇒ (unit,'σ)MON_SE
**where** \[ \text{assume-SE P = (λσ. if } \exists σ . P σ \text{ then Some((), SOME } σ . P σ \text{) else None)} \]
**notation** assume-SE (assume_SE)

**definition** if-SE :: ['σ ⇒ bool, ('α,'σ)MON_SE, ('α,'σ)MON_SE] ⇒ ('α,'σ)MON_SE
**where** \[ \text{if-SE } c E F = (λσ. if } c σ \text{ then E } σ \text{ else F } σ \]
**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)
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 \( \text{op}_1, \text{op}_2, \ldots, \text{op}_n \) with the inputs \( \iota_1, \iota_2, \ldots, \iota_n \) (outputs are treated analogously). Then we can encode for this interface the general input-type:

\[
\text{datatype in } = \text{op}_1 :: \iota_1 | \ldots | \iota_n
\]

Obviously, we lose 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 \( \text{op}_1, \text{op}_2, \ldots, \text{op}_n \) with the inputs \( \iota_1, \iota_2, \ldots, \iota_n \) (outputs are treated analogously). Then we can encode for this interface the general input-type:

\[
\text{datatype in } = \text{op}_1 :: \iota_1 | \ldots | \iota_n
\]

Obviously, we lose 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 ...

\[
\text{fun } \text{mbind} :: \text{\'t list } \Rightarrow (\text{\'t } \Rightarrow (\text{\'o, } \sigma) \text{ MON}_{SE}) \Rightarrow (\text{\'o list, } \sigma) \text{ MON}_{SE}
\]
where \( \text{mbind} \[] \text{iostep} \sigma = \text{Some}([], \sigma) \ |
\[
\begin{align*}
\text{mbind} (a # H) \text{iostep} \sigma &= \\
&= \text{(case iostep a } \sigma \text{ of} \\
&\text{ None } \Rightarrow \text{Some}([], \sigma) \ |
&\text{ Some } (\text{out}, \sigma') \Rightarrow (\text{case mbind } H \text{iostep } \sigma' \text{ of} \\
&\text{ None } \Rightarrow \text{Some}(\text{out}, \sigma') \\
&\text{ Some } (\text{outs}, \sigma'') \Rightarrow \text{Some}(\text{out} \# \text{outs}, \sigma''))
\end{align*}
\]

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

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

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

**Fun** \( \text{mbind}' : \text{'list } \Rightarrow ('o \Rightarrow ('o,'o) \text{MON}_{SE} ) \Rightarrow ('o \text{ list}, \sigma) \text{ MON}_{SE} \)

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

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

**Where**

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

In contrast \( \text{mbind} \) as a failure safe operator can roughly be seen as a \( \text{foldr} \) on \( \text{bind} - \text{try}: m1 ; m2 ; m3 ; \ldots \). Note, that the rough equivalence only holds for certain predicates in the sequence - length equivalence modulo \text{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: bind-SE-def try-SE-def)

**Subgoal for** \( x \)

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

**Apply** (simp)

**Apply** (safe)[1]

**Apply** (simp add: bind-SE-def try-SE-def)

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

**Apply** (case-tac 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** \( \text{alt-SE} :: [(\text{'o}, 'σ)\text{MON}_{SE}, (\text{'o}, 'σ)\text{MON}_{SE}] \Rightarrow (\text{'o}, 'σ)\text{MON}_{SE} \) (infixl \( \cap_{SE} \ 10 \))

**Where**

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

**Definition** \( \text{malt-SE} :: (\text{'o}, 'σ)\text{MON}_{SE} \text{ list } \Rightarrow (\text{'o}, 'σ)\text{MON}_{SE} \)

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

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

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

**By** (simp add: malt-SE-def)

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

69
**by(simp add: malt-SE-def)**

**State-Backtrack Monads**

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

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

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

**where** \( \text{bind-SB} f g \sigma = \bigcup ((\lambda(out, \sigma). (g out \sigma)) \circ (f \sigma)) \)

**notation** \( \text{bind-SB} (\text{bind}_{SB}) \)

**definition** \( \text{unit-SB} :: (\sigma) \text{MON}_{SB} = \sigma \Rightarrow (\sigma) \text{MON}_{SB} \)

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

**notation** \( \text{unit-SB} (\text{unit}_{SB}) \)

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

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

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

**apply** \( \text{(rule ext)} \)

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

**done**

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

**apply** \( \text{(rule ext)} \)

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

**done**

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

**apply** \( \text{(rule ext)} \)

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

**done**

**State Backtrack Exception Monad**

The following combination of the previous two Monad-Constructions allows for the semantic foundation of a simple generic assertion language in the style of Schirmer’s Simpl-Language or Rustan Leino’s Boogie-PL language. The key is to use the exceptional element None for violations of the assert-statement.
type-synonym \((\alpha, \sigma) \text{MON}_{\text{SBE}} = \sigma \Rightarrow ((\alpha \times \sigma) \text{set}) \text{ option}\)

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

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

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

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

definition unit-SBE :: \((\alpha) \Rightarrow (\alpha, \sigma) \text{MON}_{\text{SBE}}\) ((returning -) \(8\))

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

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

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

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

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

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

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

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

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

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

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.ccontr)
apply (simp add: Set.image-iff)
lemmas aux = trans[OF HOL.\textit{neq-commute}, OF \textit{Option.not-None-eq}]

lemma bind-assoc-SBE: \( y \equiv (x \equiv m; k); h) = (x \equiv m; (y \equiv k; h)) \)

proof (rule ext, simp add: unit-SBE-def bind-SBE-def, rename-tac x,
  case-tac m x, simp-all add: Let-def Set.image-iff, safe,goal-cases)
  case (1 x a aa b ab ba a b)
  then show ?case by (rule-tac x = (a, b) in bexI, simp-all)
next
  case (2 x a aa b ab ba)
  then show ?case 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 aa b)
  then show ?case by (erule-tac Q = None = X for X in contrapos-pp)
  apply (erule-tac x = (aa, b) in bexI, simp-all add: split-def)
  apply (erule-tac x = (aa, b) in ballE)
  apply (auto simp: aux image-def split-def intro!: rev-bexI)
  done
next
  case (5 x a aa b ab ba a b)
  then show ?case apply simp apply ((erule-tac x = (ab, ba) in ballE)+)
    apply (simp-all add: aux, (erule exE)+, simp add: split-def)
    apply (erule rev-bexI, case-tac None \( \in \) (\( \lambda \) p. h(snd p))') 'y, auto simp:split-def)
    done
next
  case (6 x a aa b a b)
  then show ?case apply simp apply ((erule-tac x = (a, b) in ballE)+)
    apply (simp-all add: aux, (erule exE)+, simp add: split-def)
    apply (erule rev-bexI, case-tac None \( \in \) (\( \lambda \) p. h(snd p))') 'y, auto simp:split-def)
    done
qed
5.1.2 Valid Test Sequences in the State Exception Monad

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

**definition** valid-SE :: \( '\sigma \Rightarrow (\text{bool},'\sigma) \) MON\( _{SE} \Rightarrow \) bool (infix \( \mid = \))

**where** \((\sigma \mid = m) = (m \sigma \neq \text{None} \land \text{fst}((\text{m } \sigma)))\)

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

**lemma syntax-test :**
\(\sigma \mid = (os \leftarrow (\text{mbind } i s \text{ ioprog}); \text{return}(\text{length } i s = \text{length } os))\)

**oops**

**lemma valid-true [simp]:** \((\sigma \mid = (s \leftarrow \text{return } x ; \text{return } (P s))) = P x\)

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

Recall \text{mbind} \_unit for the base case.

**lemma valid-failure: ioprog a \sigma = \text{None} \implies\**
\((\sigma \mid = (s \leftarrow \text{mbind} (a\#S) \text{ ioprog } ; M s)) = (\sigma \mid = (M []))\)

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

**lemma valid-failure': A \sigma = \text{None} \implies\**
\((\sigma \mid = ((s \leftarrow A ; M s)))\)

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

**lemma valid-successElem:**
\(M \sigma = \text{Some}(f \sigma,\sigma) \implies (\sigma \mid = M) = f \sigma\)

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

**lemma valid-success: ioprog a \sigma = \text{Some}(b,\sigma') \implies\**
\((\sigma \mid = (s \leftarrow \text{mbind} (a\#S) \text{ ioprog } ; M s)) = (\sigma' \mid = (s \leftarrow \text{mbind} S \text{ ioprog } ; M (b\#s)))\)

apply (simp add: valid-SE-def unit-SE-def bind-SE-def)
apply (cases \text{mbind} S \text{ ioprog} \sigma', auto)
done

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

lemma valid-success': A σ = Some(b,σ') =⇒ (σ |= (s ← A ; M s)) = (σ' |= (M b))
by (simp add: valid-SE-def unit-SE-def bind-SE-def)

lemma valid-both: (σ |= (s ← mbind (a#S) iprog ; return (P s))) =
(case iprog a σ of
None ⇒ (σ |= (return (P [])))
| Some(b,σ') ⇒ (σ' |= (s ← mbind S iprog ; return (P (b#s)))))
apply (case-tac iprog a σ)
apply (simp-all add: valid-failure valid-success'' split: prod.splits)
done

lemma valid-propagate-1 [simp]: (σ |= (return P)) = (P)
by (auto simp: valid-SE-def unit-SE-def)

lemma valid-propagate-2: σ |= ((s ← A ; M s)) =⇒ ∃ v v'. the(A σ) = (v,v') ∧ σ' |= (M v)
apply (auto simp: valid-SE-def unit-SE-def bind-SE-def)
apply (cases A σ)
apply (simp-all)
apply (drule-tac x=A σ and f=the in arg-cong)
apply (simp)
apply (rename-tac a b aa)
apply (rule-tac x=fst aa in exI)
apply (rule-tac x=snd aa in exI)
by (auto)

lemma valid-propagate-2': σ |= ((s ← A ; M s)) =⇒ ∃ a. (A σ) = Some a ∧ (snd a) |= (M (fst a))
apply (auto simp: valid-SE-def unit-SE-def bind-SE-def)
apply (cases A σ)
apply (simp-all)
apply (simp-all split: prod.splits)
apply (drule-tac x=A σ and f=the in arg-cong)
apply (simp)
apply (rename-tac a b aa x1 x2)
apply (rule-tac x=fst aa in exI)
apply (rule-tac x=snd aa in exI)
apply (auto)
done

**Lemma valid-propagate-2**: \( \sigma \models (s \leftarrow A ; M s) \implies \exists u \sigma'. A \sigma = \text{Some}(v, \sigma') \land \sigma' \models (M u) \)
- apply (auto simp: valid-SE-def unit-SE-def bind-SE-def)
- apply (cases \( \sigma \))
- 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 \( x = \text{fst} aa \) in exI)
- apply (rule-tac \( x = \text{snd} aa \) in exI)
- apply (auto)
done

**Lemma valid-propagate-3**: \((\sigma_0 \models (\lambda \sigma. \text{Some}(f \sigma, \sigma))) = (f \sigma_0)\)
- by (simp add: valid-SE-def)

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

**Lemma assert-disch1**: \(P \sigma \implies (\sigma \models (x \leftarrow \text{assert}_{SE} P ; M x)) = (\sigma \models (M \text{True}))\)
- by (auto simp: bind-SE-def assert-SE-def valid-SE-def split: HOL.if-split-asm)

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

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

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

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

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

\[
\text{lemma if-SE-D1}: \text{P}\ \sigma \implies (\sigma \models \text{if}_S E\ \text{P}\ \text{B}_1\ \text{B}_2) = (\sigma \models \text{B}_1)
\]

\[
\text{by}(auto\ simp: \text{if-SE-def\ valid-SE-def})
\]

\[
\text{lemma if-SE-D2}: \neg \text{P}\ \sigma \implies (\sigma \models \text{if}_S E\ \text{P}\ \text{B}_1\ \text{B}_2) = (\sigma \models \text{B}_2)
\]

\[
\text{by}(auto\ simp: \text{if-SE-def\ valid-SE-def})
\]

\[
\text{lemma if-SE-split-asm}: (\sigma \models \text{if}_S E\ \text{P}\ \text{B}_1\ \text{B}_2) = ((\text{P}\ \sigma \wedge (\sigma \models \text{B}_1)) \lor (\neg \text{P}\ \sigma \wedge (\sigma \models \text{B}_2)))
\]

\[
\text{by(cases\ P\ \sigma,\ auto\ simp: \text{if-SE-D1\ if-SE-D2})}
\]

\[
\text{lemma if-SE-split}: (\sigma \models \text{if}_S E\ \text{P}\ \text{B}_1\ \text{B}_2) = ((\text{P}\ \sigma \rightarrow (\sigma \models \text{B}_1)) \land (\neg \text{P}\ \sigma \rightarrow (\sigma \models \text{B}_2)))
\]

\[
\text{by(cases\ P\ \sigma,\ auto\ simp: \text{if-SE-D1\ if-SE-D2})}
\]

\[
\text{lemma [code]}: (\sigma \models m) = (\text{case}\ (m\ \sigma)\ \text{of}\ \text{None} \Rightarrow \text{False} \mid (\text{Some}\ (x,y)) \Rightarrow x)
\]

\[
\text{apply}(simp\ add: \text{valid-SE-def})
\]

\[
\text{apply}(cases\ m\ \sigma = \text{None})
\]

\[
\text{apply}(simp-all)
\]

\[
\text{apply}(insert\ not-None-eq)
\]

\[
\text{apply}(auto)
\]

\[
\text{done}
\]

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.

\[
\text{definition}\ \text{valid-SBE}::\ \sigma \Rightarrow (a,\sigma)\ \text{MON}_{\text{SBE}} \Rightarrow \text{bool}\ \text{(infix}\models_{\text{SBE}}\ \text{15})
\]

\[
\text{where}\ \sigma \models_{\text{SBE}} m \equiv (m\ \sigma \neq \text{None})
\]

This notation considers all non-failures as valid.

\[
\text{lemma assume-assert}: (\sigma \models_{\text{SBE}} (\ - \equiv \text{assume}_{\text{SBE}}\ \text{P} : \text{assert}_{\text{SBE}}\ \text{Q})) = (\text{P}\ \sigma \rightarrow Q\ \sigma)
\]

\[
\text{by(simp\ add: valid-SBE-def\ assume-SBE-def\ assert-SBE-def\ bind-SBE-def})
\]

\[
\text{lemma assert-intro}: Q\ \sigma \implies \sigma \models_{\text{SBE}} (\text{assert}_{\text{SBE}}\ \text{Q})
\]

\[
\text{by(simp\ add: valid-SBE-def\ assume-SBE-def\ assert-SBE-def\ bind-SBE-def})
\]
end
Bibliography


