The Unified Policy Framework (UPF)

Achim D. Brucker∗ Lukas Brügger† Burkhart Wolff‡

∗SAP SE, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com

†Information Security, ETH Zurich, 8092 Zurich, Switzerland
Lukas.A.Bruegger@gmail.com

‡Univ. Paris-Sud, Laboratoire LRI, UMR8623, 91405 Orsay, France France
burkhart.wolff@lri.fr

August 16, 2018

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

1 Introduction 5

2 The Unified Policy Framework (UPF) 7
   2.1 The Core of the Unified Policy Framework (UPF)  . . . . . . . . . . . . . . 7
      2.1.1 Foundation  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
      2.1.2 Policy Constructors  . . . . . . . . . . . . . . . . . . . . . . . . . 8
      2.1.3 Override Operators . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      2.1.4 Coercion Operators  . . . . . . . . . . . . . . . . . . . . . . . . . . 11
   2.2 Elementary Policies  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
      2.2.1 The Core Policy Combinators: Allow and Deny Everything  . . . . 13
      2.2.2 Common Instances  . . . . . . . . . . . . . . . . . . . . . . . . . . 14
      2.2.3 Domain, Range, and Restrictions  . . . . . . . . . . . . . . . . . . . 15
   2.3 Sequential Composition  . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
      2.3.1 Flattening  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
      2.3.2 Policy Composition  . . . . . . . . . . . . . . . . . . . . . . . . . . 20
   2.4 Parallel Composition  . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
      2.4.1 Parallel Combinators: Foundations  . . . . . . . . . . . . . . . . . . 22
      2.4.2 Combinators for Transition Policies  . . . . . . . . . . . . . . . . . 24
      2.4.3 Range Splitting  . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
      2.4.4 Distributivity of the parallel combinators  . . . . . . . . . . . . . 25
   2.5 Properties on Policies  . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
      2.5.1 Basic Properties  . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
      2.5.2 Combined Data-Policy Refinement  . . . . . . . . . . . . . . . . . . 27
      2.5.3 Equivalence of Policies  . . . . . . . . . . . . . . . . . . . . . . . . 28
   2.6 Policy Transformations  . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
      2.6.1 Elementary Operators  . . . . . . . . . . . . . . . . . . . . . . . . . 29
      2.6.2 Distributivity of the Transformation  . . . . . . . . . . . . . . . . . 32
   2.7 Policy Transformation for Testing  . . . . . . . . . . . . . . . . . . . . . 34
   2.8 Putting Everything Together: UPF  . . . . . . . . . . . . . . . . . . . . . . 36

3 Example 37

3.1 Secure Service Specification  . . . . . . . . . . . . . . . . . . . . . . . . . 37
   3.1.1 Datatypes for Modelling Users and Roles  . . . . . . . . . . . . . . 37
   3.1.2 Modelling Health Records and the Web Service API  . . . . . . . . . 38
   3.1.3 Modelling Access Control  . . . . . . . . . . . . . . . . . . . . . . . 41
   3.1.4 The State Transitions and Output Function  . . . . . . . . . . . . . 45
   3.1.5 Combine All Parts  . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
1 Introduction

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

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

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

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

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

2.1 The Core of the Unified Policy Framework (UPF)

theory

UPFCore

imports

Monads

begin

2.1.1 Foundation

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

```plaintext
datatype \('\alpha\) decision = allow \('\alpha\) | deny \('\alpha\)

type-synonym \((\alpha,\beta)\) policy = \('\alpha\ -\to \ '\beta\) decision (infixr \|-> 0)
```

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

```plaintext
translations (type) \(\alpha\ |-> \ '\beta\ <= (type) \ '\alpha\ -\to \ '\beta\) decision

type-notation policy (infixr \mapsto 0)
```
... allowing the notation \( \alpha \mapsto \beta \) for the policy type and the alternative notations for \( \text{None} \) and \( \text{Some} \) of the HOL library \( \alpha \text{ option} \) type:

<table>
<thead>
<tr>
<th>notation</th>
<th>None (⊥)</th>
</tr>
</thead>
<tbody>
<tr>
<td>notation</td>
<td>Some ([</td>
</tr>
</tbody>
</table>

Thus, the range of a policy may consist of \( \lfloor \text{accept } x \rfloor \) data, of \( \lfloor \text{deny } x \rfloor \) data, as well as \( \bot \) modeling the undefinedness of a policy, i.e. a policy is considered as a partial function. Partial functions are used since we describe elementary policies by partial system behaviour, which are glued together by operators such as function override and functional composition.

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

**definition** \( \text{Allow} :: (\alpha \text{ decision}) \text{ set} \)

<table>
<thead>
<tr>
<th>where</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Allow} = \text{range } \text{allow} )</td>
</tr>
</tbody>
</table>

**definition** \( \text{Deny} :: (\alpha \text{ decision}) \text{ set} \)

<table>
<thead>
<tr>
<th>where</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Deny} = \text{range } \text{deny} )</td>
</tr>
</tbody>
</table>

### 2.1.2 Policy Constructors

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

\( \text{?f} (\text{?a := ?b}) = (\lambda x. \text{if } x = \text{?a then ?b else ?f } x) \) and the maplet-notation \( a (x \mapsto y) \)

used for \( a (x \mapsto y) \).

Furthermore, we add notation adopted to our problem domain:

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

<table>
<thead>
<tr>
<th>syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>( -\text{policylet1} :: [\alpha, \alpha] =&gt; \text{policylet} )</td>
</tr>
<tr>
<td>( -\text{policylet2} :: [\alpha, \alpha] =&gt; \text{policylet} )</td>
</tr>
<tr>
<td>( : \text{policylet} =&gt; \text{policylets} )</td>
</tr>
<tr>
<td>( -\text{Maplets} :: [\text{policylet}, \text{policylets}] =&gt; \text{policylets} )</td>
</tr>
<tr>
<td>( -\text{Maplets} :: [\text{policylet}, \text{policylets}] =&gt; \text{policylets} )</td>
</tr>
<tr>
<td>( : \text{MapUpd} :: [\alpha, -\alpha, \text{policylets}] =&gt; \text{policylets} )</td>
</tr>
<tr>
<td>( -\text{emptypolicy} :: \alpha =&gt; \alpha )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>translations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( -\text{MapUpd} \text{ m } (-\text{Maplets } \text{xy} \text{ ms}) \Rightarrow -\text{MapUpd} (-\text{MapUpd } \text{m xy} \text{ ms}) )</td>
</tr>
<tr>
<td>( -\text{MapUpd} \text{ m } (-\text{policylet1 } \text{x y}) \Rightarrow \text{m(x := CONST Some (CONST allow y))} )</td>
</tr>
<tr>
<td>( -\text{MapUpd} \text{ m } (-\text{policylet2 } \text{x y}) \Rightarrow \text{m(x := CONST Some (CONST deny y))} )</td>
</tr>
<tr>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \Rightarrow \text{CONST Map.empty} )</td>
</tr>
</tbody>
</table>

Here are some lemmas essentially showing syntactic equivalences:

**lemma** \( \text{test}: \emptyset(\mapsto_+a, \mapsto_- b) = \emptyset(\mapsto_+ a, \mapsto_- b) \)  \( \langle \text{proof} \rangle \)
lemma test2: \( p(x \mapsto +a, x \mapsto -b) = p(x \mapsto -b) \) \( \langle \text{proof} \rangle \)

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

lemma pol-upd-triv1: \( t \ k = \lfloor \text{allow } x \rfloor \implies t(k \mapsto +x) = t \) \( \langle \text{proof} \rangle \)

lemma pol-upd-triv2: \( t \ k = \lfloor \text{deny } x \rfloor \implies t(k \mapsto -x) = t \) \( \langle \text{proof} \rangle \)

lemma pol-upd-allow-nonempty: \( t(k \mapsto +x) \neq \emptyset \) \( \langle \text{proof} \rangle \)

lemma pol-upd-deny-nonempty: \( t(k \mapsto -x) \neq \emptyset \) \( \langle \text{proof} \rangle \)

lemma pol-upd-eqD1: \( m(a \mapsto +x) = n(a \mapsto +y) \implies x = y \) \( \langle \text{proof} \rangle \)

lemma pol-upd-eqD2: \( m(a \mapsto -x) = n(a \mapsto -y) \implies x = y \) \( \langle \text{proof} \rangle \)

lemma pol-upd-neq1 [simp]: \( m(a \mapsto +x) \neq n(a \mapsto -y) \) \( \langle \text{proof} \rangle \)

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
- \( \text{-policyoverride} :: ['a \mapsto 'b, 'a \mapsto 'b] \Rightarrow 'a \mapsto 'b \text{ (infixl } \oplus \text{ 100}) \)

translations
\[ p \oplus q = q ++ p \]

Some elementary facts inherited from Map are:

lemma override-empty: \( p \oplus \emptyset = p \) \( \langle \text{proof} \rangle \)

lemma empty-override: \( \emptyset \oplus p = p \) \( \langle \text{proof} \rangle \)
lemma override-assoc: \( p_1 \oplus (p_2 \oplus p_3) = (p_1 \oplus p_2) \oplus p_3 \)

(proof)

The following two operators are variants of the standard override. For \( \text{override}_A \), an allow of wins over a deny. For \( \text{override}_D \), the situation is dual.

definition override-A :: \( \left[ \alpha \mapsto \beta, \alpha \mapsto \beta \right] \Rightarrow \left[ \alpha \mapsto \beta \right] \) (infixl ++'A 100)
where \( m_2 ++_A m_1 = \)

\( \lambda x. \text{case } m_1 x \text{ of} \)
\( \text{[allow } a \text{] } \Rightarrow \text{[allow } a \text{]} \)
\( \text{[deny } a \text{]} \Rightarrow \text{(case } m_2 x \text{ of} \)
\( \text{[allow } b \text{] } \Rightarrow \text{[allow } b \text{]} \)
\( \text{- } \Rightarrow \text{[deny } a \text{]} \)
\( \text{| } \perp \Rightarrow m_2 x \)

)

syntax
-\( \text{policyoverride-A} :: \left[ \alpha \mapsto \beta, \alpha \mapsto \beta \right] \Rightarrow \left[ \alpha \mapsto \beta \right] \) (infixl \( \bigoplus \) A 100)
translations
\( p \bigoplus_A q \Leftarrow p ++_A q \)

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

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

lemma override-A-assoc: \( p_1 \bigoplus_A (p_2 \bigoplus_A p_3) = (p_1 \bigoplus_A p_2) \bigoplus_A p_3 \)
(proof)

definition override-D :: \( \left[ \alpha \mapsto \beta, \alpha \mapsto \beta \right] \Rightarrow \left[ \alpha \mapsto \beta \right] \) (infixl ++'D 100)
where \( m_1 +++_D m_2 = \)

\( \lambda x. \text{case } m_2 x \text{ of} \)
\( \text{[deny } a \text{] } \Rightarrow \text{[deny } a \text{]} \)
\( \text{[allow } a \text{]} \Rightarrow \text{(case } m_1 x \text{ of} \)
\( \text{[deny } b \text{]} \Rightarrow \text{[deny } b \text{]} \)
\( \text{- } \Rightarrow \text{[allow } a \text{]} \)
\( \text{| } \perp \Rightarrow m_1 x \)

syntex
-\( \text{policyoverride-D} :: \left[ \alpha \mapsto \beta, \alpha \mapsto \beta \right] \Rightarrow \left[ \alpha \mapsto \beta \right] \) (infixl \( \bigoplus \) D 100)
translations
\( p \bigoplus_D q \Leftarrow p +++_D q \)

lemma override-D-empty[simp]: \( p \bigoplus_D \emptyset = p \)
proof

lemma empty-override-D[simp]: $\emptyset \oplus_{D} p = p$

lemma override-D-assoc: $p_1 \oplus_{D} (p_2 \oplus_{D} p_3) = (p_1 \oplus_{D} p_2) \oplus_{D} p_3$

proof

2.1.4 Coercion Operators

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

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

definition policy-range-comp :: \([\beta \Rightarrow \gamma], \alpha \mapsto \beta \Rightarrow \alpha \mapsto \gamma\) (infixl o⁻f 55)

where

\[
\begin{align*}
    f \circ_f p &= (\lambda x. \text{case } p x \text{ of} \\
    & \quad \allow y \Rightarrow \allow (f y) \\
    & \quad \deny y \Rightarrow \deny (f y) \\
    & \quad \bot \Rightarrow \bot
\end{align*}
\]

syntax

policy-range-comp :: \([\beta \Rightarrow \gamma], \alpha \mapsto \beta \Rightarrow \alpha \mapsto \gamma\) (infixl o_f 55)

translations

\[ p \circ_f q \equiv p \circ_f q \]

lemma policy-range-comp-strict : $f \circ_f \emptyset = \emptyset$

proof

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 :: \([\beta \Rightarrow \gamma], \alpha \mapsto \beta \Rightarrow \alpha \mapsto \gamma\) (infixr ∇ 100)

where

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

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

proof

lemma range-split-charn:

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

\[ \]
allow x ⇒ [allow (f x)]
| deny x ⇒ [deny (g x)]
| ⊥ ⇒ ⊥)

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

lemma range-split-vs-range-compose: (f . f) ∇ p = f o f p
⟨proof⟩

lemma range-split-id [simp]: (id . id) ∇ p = p
⟨proof⟩

lemma range-split-bi-compose [simp]: (f1 . f2) ∇ (g1 . g2) ∇ p = (f1 o g1 . f2 o g2) ∇ p
⟨proof⟩
The next three operators are rather exotic and in most cases not used.
The following is a variant of range_split, where the change in the decision depends on the input instead of the output.

definition dom-split2a :: [′α ⇀′ γ] × [′α ⇀′ γ] ⇒′ α ↦′ γ (infixr ∆ 100)
where P ∆ a p = (λx. case p x of
| allow y ⇒ allow (the ((fst P) x))]
| deny y ⇒ deny (the ((snd P) x))]
| ⊥ ⇒ ⊥)

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

definition range-split2 :: [(′α ⇒′ γ) × (′α ⇒′ γ)] ⇒′ α ⇒′ γ (infixr ∇ 100)
where P ∇ 2 p = (λx. case p x of
| allow y ⇒ allow (y,(fst P) x)]
| deny y ⇒ deny (y,(snd P) x)]
| ⊥ ⇒ ⊥)

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

definition policy2MON :: (′i×′σ ⇒′ o×′σ) ⇒ (′i ⇒(′o decision,′σ) MON_SE)
where $\text{policy2MON } p = (\lambda \ i \ \sigma. \ \text{case } p \ (i,\sigma) \ of$
\[
\begin{align*}
&\text{[(allow (outs,}\sigma'))] \Rightarrow \text{[(allow outs, } \sigma')]} \\
&\mid \text{[(deny (outs,}\sigma'))] \Rightarrow \text{[(deny outs, } \sigma')]} \\
&\mid \bot \Rightarrow \bot
\end{align*}
\]

lemmas $\text{UPFCoreDefs } = \text{Allow-def Deny-def override-A-def override-D-def}$
$policy\text{-range-comp-def}$
$\text{range-split-def dom-split2-def map-add-def restrict-map-def}$

end

2.2 Elementary Policies

theory $\text{ElementaryPolicies}$
imports $\text{UPFCore}$
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 built by composing the elementary policies defined in this theory.

2.2.1 The Core Policy Combinators: Allow and Deny Everything

definition $\text{deny-pfun} :: ((\alpha \rightarrow \beta) \Rightarrow (\alpha \rightarrow \beta) \ (\text{AllD})$ where
$\text{deny-pfun } pf \equiv (\lambda \ x. \ \text{case } pf \ x \ of$
\[
\begin{align*}
&\text{[(y)] } \Rightarrow \text{[(deny (y)]} \\
&\bot \Rightarrow \bot
\end{align*}
\]
definition $\text{allow-pfun} :: ((\alpha \rightarrow \beta) \Rightarrow (\alpha \rightarrow \beta) \ (\text{AllA})$ where
$\text{allow-pfun } pf \equiv (\lambda \ x. \ \text{case } pf \ x \ of$
\[
\begin{align*}
&\text{[(y)] } \Rightarrow \text{[(allow (y)]} \\
&\bot \Rightarrow \bot
\end{align*}
\]
syntax $\text{-allow-pfun} :: ((\alpha \rightarrow \beta) \Rightarrow (\alpha \rightarrow \beta) \ (A_p)$
translations $A_p \ f = \text{AllA } f$
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 ⌋)
  ⟨proof⟩

lemma AllD-norm2[simp]: deny-pfun (Some o id) = (∀ Dx. ⌊ x ⌋)
  ⟨proof⟩

lemma AllA-norm[simp]: allow-pfun (id o Some) = (∀ Ax. ⌊ x ⌋)
  ⟨proof⟩

lemma AllA-norm2[simp]: allow-pfun (Some o id) = (∀ Ax. ⌊ x ⌋)
  ⟨proof⟩

lemma AllA-apply[simp]: (∀ Ax. Some (P x)) x = ⌊ allow (P x) ⌋
  ⟨proof⟩

lemma AllD-apply[simp]: (∀ Dx. Some (P x)) x = ⌊ deny (P x) ⌋
  ⟨proof⟩

lemma neq-Allow-Deny: pf ≠ ∅ =⇒ (deny-pfun pf) ≠ (allow-pfun pf)
  ⟨proof⟩

2.2.2 Common Instances

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

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

definition
  deny-all-id :: 'α ⇒ 'α (D_f) where
  deny-all-id ≜ deny-pfun (id o Some)

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

definition  
allow-all :: (α → unit) (A_U) where
allow-all p = [allow ()]

definition  
deny-all :: (α → unit) (D_U) where
deny-all p = [deny ()]

... and resulting properties:

lemma A_I ⊕ Map.empty = A_I
〈proof〉

lemma A_f f ⊕ Map.empty = A_f f
〈proof〉

lemma allow-pfun Map.empty = Map.empty
〈proof〉

lemma allow-left-cancel : dom pf = UNIV → (allow-pfun pf) ⊕ x = (allow-pfun pf)
〈proof〉

lemma deny-left-cancel : dom pf = UNIV → (deny-pfun pf) ⊕ x = (deny-pfun pf)
〈proof〉

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 ≠ ⊥ \}

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 ∈ dom ?m → ∃ b. ?m ?a = [b]
- Map.domI ?m ?a = [?b] → ?a ∈ dom ?m
- Map.domIff (?a ∈ dom ?m) = (?m ?a ≠ ⊥)
• Map.dom_const \( \text{dom} \ (\lambda x. \lfloor f x \rfloor) = \text{UNIV} \)

• Map.dom_def \( \text{dom} \ ?m = \{ a. \ ?m a \neq \bot \} \)

• Map.dom_empty \( \text{dom} \ \emptyset = \{ \} \)

• Map.dom_eq_empty_conv \( \text{dom} \ ?f = \{ \} \) = \( \{ f = \emptyset \} \)

• Map.dom_eq_singleton_conv \( \text{dom} \ ?f = \{ x \} \) = \( \{ \exists v. \ ?f = [x \mapsto v] \} \)

• Map.dom_fun_upd \( \text{dom} \ (\ ?f (?x := ?y)) \) = \( \{ \text{if } ?y = \bot \text{ then } \text{dom} \ ?f - \{ ?x \} \text{ else } \text{insert } ?x \ (\text{dom } \ ?f) \} \)

• Map.dom_if \( \text{dom} \ (\lambda x. \text{if } ?P x \text{ then } ?f x \text{ else } ?g x) = \text{dom } \ ?f \cap \{ x. \ ?P x \} \cup \text{dom } \ ?g \cap \{ x. \neg ?P x \} \)

• Map.dom_map_add \( \text{dom} \ (\ ?n \oplus \ ?m) = \text{dom } \ ?n \cup \text{dom } \ ?m \)

However, some properties are specific to policy concepts:

**lemma** sub-ran : \( \text{ran} \ p \subseteq \text{Allow} \cup \text{Deny} \)

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

**lemma** dom-alisa-all: \( \text{dom}(\text{A} \ f \ f) = \text{UNIV} \)

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

**lemma** dom-deny-all: \( \text{dom}(\text{D} \ f \ id) = \text{UNIV} \)

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

**lemma** ran-alisa-all: \( \text{ran}(\text{A} \ f \ id) = \text{Allow} \)

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

**lemma** ran-deny-all: \( \text{ran}(\text{D} \ f \ id) = \text{Deny} \)
Reasoning over \texttt{dom} is most crucial since it paves the way for simplification and reordering of policies composed by override (i.e. by the normal left-to-right rule composition method.

- \texttt{Map.dom_map_add} \texttt{dom (?n \ominus ?m) = dom ?n \cup dom ?m}
- \texttt{Map.inj_on_map_add} \texttt{inj-on (?m' \ominus ?m) (dom ?m') = inj-on ?m' (dom ?m')}
- \texttt{Map.map_add_comm} \texttt{dom ?m1.0 \cap dom ?m2.0 = \{\} \Rightarrow ?m2.0 \oplus ?m1.0 = ?m1.0 \oplus ?m2.0}
- \texttt{Map.map_add_dom_app_simps(1)} \texttt{?m \in dom ?l2.0 = \Rightarrow (?l2.0 \oplus ?l1.0) ?m = ?l2.0 ?m}
- \texttt{Map.map_add_dom_app_simps(2)} \texttt{?m \notin dom ?l1.0 = \Rightarrow (?l2.0 \oplus ?l1.0) ?m = ?l1.0 ?m}
- \texttt{Map.map_add_dom_app_simps(3)} \texttt{?m \notin dom ?l2.0 = \Rightarrow (?l2.0 \oplus ?l1.0) ?m = ?l1.0 ?m}
- \texttt{Map.map_add_upd_left} \texttt{?m \notin dom ?e2.0 = \Rightarrow ?e2.0 \oplus ?e1.0(?m \mapsto ?u1.0) = (?e2.0 \oplus ?e1.0)(?m \mapsto ?u1.0)}

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

\textbf{definition} \texttt{dom-restrict :: ['\alpha set, '\alpha \rightarrow \beta] \Rightarrow '\alpha \rightarrow \beta (infixr \triangleq 55)}

\textbf{where} \texttt{S \triangleq (\lambda x. if x \in S then p x else \bot)}

\textbf{lemma} \texttt{dom-dom-restrict[simp] : dom(S \triangleq p) = S \cap dom p}

\textbf{lemma} \texttt{dom-restrict-idem[simp] : (dom p) \triangleq p = p}

\textbf{lemma} \texttt{dom-restrict-inter[simp] : T \triangleq S \triangleq p = T \cap S \triangleq p}

\textbf{definition} \texttt{ran-restrict :: ['\alpha \rightarrow \beta, \beta decision set] \Rightarrow '\alpha \rightarrow \beta (infixr \triangleright 55)}

\textbf{where} \texttt{p \triangleright S \equiv (\lambda x. if p x \in (Some'S) then p x else \bot)}

\textbf{definition} \texttt{ran-restrict2 :: ['\alpha \rightarrow \beta, \beta decision set] \Rightarrow '\alpha \rightarrow '\beta (infixr \triangleright 2 55)}

\textbf{where} \texttt{p \triangleright\triangleright 2 S \equiv (\lambda x. if (the (p x)) \in (S) then p x else \bot)}

\textbf{lemma} \texttt{ran-restrict = ran-restrict2
2.3 Sequential Composition

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
\[ \text{flat-orA}(\text{deny}(\text{allow } y)) = \text{allow } y \]
\[ \text{flat-orA}(\text{deny}(\text{deny } y)) = \text{deny } y \]

**Lemma** flat-orA-deny[dest]: flat-orA \( x = \text{deny } y \Rightarrow x = \text{deny}(\text{deny } y) \)

\{proof\}

**Lemma** flat-orA-allow[dest]: flat-orA \( x = \text{allow } y \Rightarrow x = \text{allow}(\text{allow } y) \)
\[ \lor x = \text{allow}(\text{deny } y) \]
\[ \lor x = \text{deny}(\text{allow } y) \]

\{proof\}

**Fun** flat-orD :: ('\( \alpha \) decision) decision \( \Rightarrow \) ('\( \alpha \) decision)

**Where**
\[ \text{flat-orD}(\text{allow}(\text{allow } y)) = \text{allow } y \]
\[ \text{flat-orD}(\text{deny}(\text{deny } y)) = \text{deny } y \]
\[ \text{flat-orD}(\text{deny}(\text{allow } y)) = \text{deny } y \]
\[ \text{flat-orD}(\text{allow}(\text{deny } y)) = \text{deny } y \]

**Lemma** flat-orD-allow[dest]: flat-orD \( x = \text{allow } y \Rightarrow x = \text{allow}(\text{allow } y) \)

\{proof\}

**Lemma** flat-orD-deny[dest]: flat-orD \( x = \text{deny } y \Rightarrow x = \text{deny}(\text{deny } y) \)
\[ \lor x = \text{allow}(\text{deny } y) \]
\[ \lor x = \text{deny}(\text{allow } y) \]

\{proof\}

**Fun** flat-1 :: ('\( \alpha \) decision) decision \( \Rightarrow \) ('\( \alpha \) decision)

**Where**
\[ \text{flat-1}(\text{allow}(\text{allow } y)) = \text{allow } y \]
\[ \text{flat-1}(\text{deny}(\text{deny } y)) = \text{den}\]
\[ \text{flat-1}(\text{deny}(\text{allow } y)) = \text{allow } y \]
\[ \text{flat-1}(\text{allow}(\text{deny } y)) = \text{deny } y \]

**Lemma** flat-1-allow[dest]: flat-1 \( x = \text{allow } y \Rightarrow x = \text{allow}(\text{allow } y) \lor x = \text{allow}(\text{deny } y) \)

\{proof\}

**Lemma** flat-1-deny[dest]: flat-1 \( x = \text{deny } y \Rightarrow x = \text{deny}(\text{deny } y) \lor x = \text{deny}(\text{allow } y) \)

\{proof\}

**Fun** flat-2 :: ('\( \alpha \) decision) decision \( \Rightarrow \) ('\( \alpha \) decision)

**Where**
\[ \text{flat-2}(\text{allow}(\text{allow } y)) = \text{allow } y \]
\[ \text{flat-2}(\text{deny}(\text{deny } y)) = \text{deny } y \]
\[ \text{flat-2}(\text{deny}(\text{allow } y)) = \text{allow } y \]
\[ \text{flat-2}(\text{allow}(\text{deny } y)) = \text{deny } y \]
\textbf{lemma} flat-2-allow[dest]: flat-2 \(x = \text{allow } y \implies x = \text{allow}(\text{allow } y) \lor x = \text{deny}(\text{allow } y)\)
\begin{proof}
\end{proof}

\textbf{lemma} flat-2-deny[dest]: flat-2 \(x = \text{deny } y \implies x = \text{deny}(\text{deny } y) \lor x = \text{allow}(\text{deny } y)\)
\begin{proof}
\end{proof}

\subsection*{2.3.2 Policy Composition}

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

\textbf{fun} lift :: \((\alpha \to \beta) \to (\alpha \, \text{decision} \to \beta \, \text{decision})\)
\begin{proof}
\begin{align*}
\text{lift } f \, (\text{deny } s) & = \text{case } f \, s \text{ of} \\
& \quad \{ y \Rightarrow \text{deny } y \} \\
& \quad \mid \bot \Rightarrow \bot \\
\text{lift } f \, (\text{allow } s) & = \text{case } f \, s \text{ of} \\
& \quad \{ y \Rightarrow \text{allow } y \} \\
& \quad \mid \bot \Rightarrow \bot
\end{align*}
\end{proof}

\textbf{lemma} lift-mt [simp]: lift \(\emptyset = \emptyset\)
\begin{proof}
\end{proof}

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:

\textbf{definition} comp-orA :: \([\beta \to \gamma, \alpha \to \beta] \to [\alpha \to \gamma] \, \text{infixl } o^-\alpha \, \text{orA } 55\) \begin{proof} \end{proof}

\textbf{notation} \(comp-orA \quad \text{infixl } o^-\wedge 55\)
\begin{proof}
\end{proof}

\textbf{lemma} comp-orA-mt[simp]: \(p \circ^\wedge \emptyset = \emptyset\)
\begin{proof}
\end{proof}

\textbf{lemma} mt-comp-orA[simp]: \(\emptyset \circ^\wedge p = \emptyset\)
\begin{proof}
\end{proof}

\textbf{definition} comp-orD :: \([\beta \to \gamma, \alpha \to \beta] \to [\alpha \to \gamma] \, \text{infixl } o^-\alpha \, \text{orD } 55\) \begin{proof} \end{proof}

\textbf{notation} \(comp-orD \quad \text{infixl } o^-\vee 55\)
\begin{proof}
\end{proof}

\textbf{lemma} comp-orD-mt[simp]: \(p \circ^\vee \emptyset = \emptyset\)
\begin{proof}
\end{proof}

\textbf{lemma} mt-comp-orD[simp]: \(\emptyset \circ^\vee p = \emptyset\)
\begin{proof}
\end{proof}
notation
  \text{comp-orD} \ (\text{infixl} \circ_{oR} 55)

lemma \text{comp-orD-mt}[\text{simp}]: \text{p \ o-orD} \emptyset = \emptyset
  \langle \text{proof} \rangle

lemma \text{mt-comp-orD}[\text{simp}]: \emptyset \ o-orD \text{p} = \emptyset
  \langle \text{proof} \rangle

definition
  \text{comp-1} :: \{\beta \mapsto \gamma, \alpha \mapsto \beta\} \Rightarrow \alpha \mapsto \gamma \ (\text{infixl} \circ_{\text{1}} 55) \text{ where}
  \text{p2 o-1 p1} \equiv (\text{map-option flat-1}) \circ (\text{lift p2} \circ_{m} \text{p1})

notation
  \text{comp-1} \ (\text{infixl} \circ_{1} 55)

lemma \text{comp-1-mt}[\text{simp}]: \text{p} \circ_{1} \emptyset = \emptyset
  \langle \text{proof} \rangle

lemma \text{mt-comp-1}[\text{simp}]: \emptyset \circ_{1} \text{p} = \emptyset
  \langle \text{proof} \rangle

definition
  \text{comp-2} :: \{\beta \mapsto \gamma, \alpha \mapsto \beta\} \Rightarrow \alpha \mapsto \gamma \ (\text{infixl} \circ_{\text{2}} 55) \text{ where}
  \text{p2 o-2 p1} \equiv (\text{map-option flat-2}) \circ (\text{lift p2} \circ_{m} \text{p1})

notation
  \text{comp-2} \ (\text{infixl} \circ_{2} 55)

lemma \text{comp-2-mt}[\text{simp}]: \text{p} \circ_{2} \emptyset = \emptyset
  \langle \text{proof} \rangle

lemma \text{mt-comp-2}[\text{simp}]: \emptyset \circ_{2} \text{p} = \emptyset
  \langle \text{proof} \rangle

de\text{nd}

2.4 Parallel Composition

d\text{e}\text{dteeny}
  \text{ParallelComposition}

imports
  \text{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** :: $\forall \alpha \mapsto \forall \beta \left( \forall \gamma \mapsto \forall \delta \right)$ (infixr $\otimes_A 55$)

where $p1 \otimes_A p2 =
(\lambda (x, y). \text{case } p1 x \text{ of}
 | \text{allow } d1 \Rightarrow \text{case } p2 y \text{ of}
 | \text{allow } d2 \Rightarrow \text{allow}(d1,d2)
 | \bot \Rightarrow \bot)
 | \text{deny } d1 \Rightarrow \text{case } p2 y \text{ of}
 | \text{allow } d2 \Rightarrow \text{allow}(d1,d2)
 | \bot \Rightarrow \bot)
 | \bot \Rightarrow \bot))

**lemma prod-orA-mt[simp]:** $p \otimes_A \emptyset = \emptyset$

(proof)

**lemma mt-prod-orA[simp]:** $\emptyset \otimes_A p = \emptyset$

(proof)

**lemma prod-orA-quasi-commute:** $p2 \otimes_A p1 = ((\lambda (x, y). (y, x)) \circ-f (p1 \otimes_A p2))$ $o (\lambda (a, b). (b, a))$

(proof)

**definition prod-orD** :: $\forall \alpha \mapsto \forall \beta \left( \forall \gamma \mapsto \forall \delta \right)$ (infixr $\otimes_D 55$)

where $p1 \otimes_D p2 =
(\lambda (x, y). \text{case } p1 x \text{ of}
 | \text{allow } d1 \Rightarrow \text{case } p2 y \text{ of}
 | \text{allow } d2 \Rightarrow \text{allow}(d1,d2)
 | \bot \Rightarrow \bot)
 | \text{deny } d1 \Rightarrow \text{case } p2 y \text{ of}
 | \text{allow } d2 \Rightarrow \text{allow}(d1,d2)
 | \bot \Rightarrow \bot)
 | \bot \Rightarrow \bot))$
\[
\begin{align*}
[\text{allow } d2] & \Rightarrow [\text{allow}(d1,d2)] \\
[\text{deny } d2] & \Rightarrow [\text{deny}(d1,d2)] \\
\bot & \Rightarrow \bot
\end{align*}
\]

| \[\text{deny } d1 \Rightarrow (\text{case } p2 \ y \ of \]
| \[\text{allow } d2] & \Rightarrow [\text{deny}(d1,d2)] \\
| \[\text{deny } d2] & \Rightarrow [\text{deny}(d1,d2)] \\
\bot & \Rightarrow \bot
\]

| \[\bot] & \Rightarrow \bot
\]

\textbf{lemma} prod-orD-mt[simp]: \(p \otimes_D \emptyset = \emptyset\)
\langle proof \rangle

\textbf{lemma} mt-prod-orD[simp]: \(\emptyset \otimes_D p = \emptyset\)
\langle proof \rangle

\textbf{lemma} prod-orD-quasi-commute: \(p2 \otimes_D p1 = (((\lambda(x,y). (y,x)) \ o-f (p1 \otimes_D p2)) \ o (\lambda(a,b).(b,a)))\)
\langle proof \rangle

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

\textbf{definition} prod-1 :: \([\alpha \to \beta, \gamma \to \delta] \Rightarrow (\alpha \times \gamma \to \beta \times \delta)\) \ (\text{infixr } \otimes_1 \ 55) \]
\textbf{where} \(p1 \otimes_1 p2 \equiv \)
\[(\lambda(x,y). (\text{case } p1 \ x \ of \]
| \[\text{allow } d1 \Rightarrow (\text{case } p2 \ y \ of \]
| \[\text{allow } d2] & \Rightarrow [\text{allow}(d1,d2)] \\
| \[\text{deny } d2] & \Rightarrow [\text{allow}(d1,d2)] \\
\bot & \Rightarrow \bot
\]
| \[\bot \Rightarrow \bot
\]

\textbf{lemma} prod-1-mt[simp]: \(p \otimes_1 \emptyset = \emptyset\)
\langle proof \rangle

\textbf{lemma} mt-prod-1[simp]: \(\emptyset \otimes_1 p = \emptyset\)
\langle proof \rangle

\textbf{definition} prod-2 :: \([\alpha \to \beta, \gamma \to \delta] \Rightarrow (\alpha \times \gamma \to \beta \times \delta)\) \ (\text{infixr } \otimes_2 \ 55) \]
\textbf{where} \(p1 \otimes_2 p2 \equiv \)
\[(\lambda(x,y). (\text{case } p1 \ x \ of \]
| \[\text{allow } d1 \Rightarrow (\text{case } p2 \ y \ of \]
| \[\text{allow } d2] & \Rightarrow [\text{allow}(d1,d2)] \\
\bot & \Rightarrow \bot
\]

23
lemma prod-2-mt[simp]: p ⊗ 2 ∅ = ∅  
⟨proof⟩

lemma mt-prod-2[simp]: ∅ ⊗ 2 p = ∅  
⟨proof⟩

definition prod-1-id :: [′α ↦→ ′β, ′α ↦→ ′γ] ⇒ (′α ↦→ ′β ×′γ) (infixr ⊗ 1I 55)  
where p ⊗ 1I q = (p ⊗ 1 q) o (λx. (x,x))  
lemma prod-1-id-mt[simp]: p ⊗ 1I ∅ = ∅  
⟨proof⟩

lemma mt-prod-1-id[simp]: ∅ ⊗ 1I p = ∅  
⟨proof⟩

definition prod-2-id :: [′α ↦→ ′β, ′α ↦→ ′γ] ⇒ (′α ↦→ ′β ×′γ) (infixr ⊗ 2I 55)  
where p ⊗ 2I q = (p ⊗ 2 q) o (λx. (x,x))  
lemma prod-2-id-mt[simp]: p ⊗ 2I ∅ = ∅  
⟨proof⟩

lemma mt-prod-2-id[simp]: ∅ ⊗ 2I p = ∅  
⟨proof⟩

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 ⊗ M 60)  
where p1 ⊗ M p2 = (λ (x,y). case p1 x of | d1 | ⇒  
(case p2 y of | d2 | ⇒ [(d1,d2)]  
| ⊥ ⇒ ⊥)  
| ⊥ ⇒ ⊥)

24
definition parallel-st :: ('i × 'σ → 'σ) ⇒ ('i × 'σ' → 'σ') ⇒
    ('i × 'σ × 'σ' → 'σ × 'σ') (infixr ⊗ S 60)

where
    p1 ⊗ S p2 = (p1 ⊗ M p2) o (λ (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 :: [((′α ↠ ′γ) × (′α ↠ ′γ)), ′d ↠ ′β] ⇒
    (′d × ′α) ↠ (′β × ′γ) (infixr ⊗ ∇ 100)

where
    P ⊗ ∇ p ≡ λ x. case p (fst x) of
        | allow y ⇒ (case ((fst P) (snd x)) of ⊥ ⇒ ⊥ | [z] ⇒ | allow (y,z))
        | deny y ⇒ (case ((snd P) (snd x)) of ⊥ ⇒ ⊥ | [z] ⇒ | deny (y,z))
        | ⊥ ⇒ ⊥

An alternative characterisation of the operator is as follows:

lemma comp-ran-split-charn:
    (f, g) ⊗ ∇ = ((((p ⊢ Allow) ⊗ ∇ (A_p f)) ⊕
    ((p ⊢ Deny) ⊗ ∇ (D_p g))))

2.4.4 Distributivity of the parallel combinators

lemma distr-or1-a: (F = F1 ⊕ F2) ⇒ ((N ⊗ 1 F) o f) =
    (((N ⊗ 1 F1) o f) ⊕ ((N ⊗ 1 F2) o f))

lemma distr-or1: (F = F1 ⊕ F2) ⇒ (g o-f ((N ⊗ 1 F) o f)) =
    ((g o-f ((N ⊗ 1 F1) o f)) ⊕ (g o-f ((N ⊗ 1 F2) o f)))

lemma distr-or2-a: (F = F1 ⊕ F2) ⇒ (N ⊗ 2 F) o f) =
    (((N ⊗ 2 F1) o f) ⊕ ((N ⊗ 2 F2) o f))

lemma distr-or2: (F = F1 ⊕ F2) ⇒ (r o-f ((N ⊗ 2 F) o f)) =
    ((r o-f ((N ⊗ 2 F1) o f)) ⊕ (r o-f ((N ⊗ 2 F2) o f)))
lemma distr-orA: \((F = F1 \oplus F2) \implies ((g \circ f ((N \otimes \vee_A F) \circ f)) = ((g \circ f ((N \otimes \vee_A F1) \circ f)) \oplus (g \circ f ((N \otimes \vee_A F2) \circ f))))\) 
\langle proof \rangle

lemma distr-orD: \((F = F1 \oplus F2) \implies ((g \circ f ((N \otimes \vee_D F) \circ f)) = ((g \circ f ((N \otimes \vee_D F1) \circ f)) \oplus (g \circ f ((N \otimes \vee_D F2) \circ f))))\) 
\langle proof \rangle

lemma coerc-assoc: \((r \circ f P) \circ d = r \circ f (P \circ d)\) 
\langle proof \rangle

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

2.5 Properties on Policies

theory
  Analysis
  imports
    ParallelComposition
    SeqComposition
begin

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

2.5.1 Basic Properties

A Policy Has no Gaps

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

Comparing Policies

Policy p is more defined than q:

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

definition strictly-more-defined :: ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b) \Rightarrow bool
where strictly-more-defined p q = (dom q \subset dom p)
Lemma strictly-more-vs-more: strictly-more-defined \( p \) \( q \implies \) more-defined \( p \) \( q \)

(proof)

Policy \( p \) is more permissive than \( q \):

definition more-permissive :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ bool (infixl \( \sqsubseteq_A \) 60)
where \( p \sqsubseteq_A q = (∀ x. \\text{case } q \ x \ of \ [allow \ y] ⇒ (∃ z. (p \ x = [allow \ z]))\)
| [deny \ y] ⇒ True
| ⊥ ⇒ True))

Lemma more-permissive-refl : \( p \sqsubseteq_A p \)
(proof)

Lemma more-permissive-trans : \( p \sqsubseteq_A p' \implies p' \sqsubseteq_A p'' \implies p \sqsubseteq_A p'' \)
(proof)

Policy \( p \) is more rejective than \( q \):

definition more-rejective :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ bool (infixl \( \sqsubseteq_D \) 60)
where \( p \sqsubseteq_D q = (∀ x. \\text{case } q \ x \ of \ [deny \ y] ⇒ (∃ z. (p \ x = [deny \ z]))\)
| [allow \ y] ⇒ True
| ⊥ ⇒ True))

Lemma more-rejective-trans : \( p \sqsubseteq_D p' \implies p' \sqsubseteq_D p'' \implies p \sqsubseteq_D p'' \)
(proof)

Lemma more-rejective-refl : \( p \sqsubseteq_D p \)
(proof)

Lemma \( A_f \ f \sqsubseteq_A p \)
(proof)

Lemma \( A_I \ sqsubseteq_A p \)
(proof)

2.5.2 Combined Data-Policy Refinement

definition policy-refinement ::
('a ⇒ 'b) ⇒ ('a' ⇒ 'a) ⇒ ('b' ⇒ 'b) ⇒ bool
(\(\leq_{\text{abs}_{a,b}}\) \(\leq_{[50,50,50]50}\))
where \( p \sqsubseteq_{\text{abs}_{a,b}} q \equiv \)
(∀ a. case p a of
\[ \bot \Rightarrow True \]
\[ |allow y] \Rightarrow (\forall a' \in \{x. abs_a x = a\}.
\exists b'. q a' = [allow b']
\land abs_b b' = y) \]
\[ |deny y] \Rightarrow (\forall a' \in \{x. abs_a x = a\}.
\exists b'. q a' = [deny b']
\land abs_b b' = y) \]

**Theorem polref-refl:** \( p \sqsubseteq id \), \( id p \)

**Lemma policy-eq:**

**Assumptions:**
- \( p \sqsubseteq_A q \)
- \( q \sqsubseteq_A p \)
- \( q \sqsubseteq_D p \)
- \( p \sqsubseteq_D q \)

**Shows:** \( dom p = dom q \)

**Proof:**

**Miscellaneous**

**Lemma dom-inter:** \([dom p \cap dom q = \{\}; p x = [y]] \Rightarrow q x = \bot \)

**Lemma dom-eq:** \( dom p \cap dom q = \{\} \Rightarrow p \uplus_A q = p \uplus_D q \)
lemma dom-split-alt-def : (f, g) ∆ p = (dom(p ⊢ Allow) ⊢ (A f)) ⊕ (dom(p ⊢ Deny) ⊢ (D f g))

end

2.6 Policy Transformations

theory Normalisation

imports SeqComposition

begin

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

2.6.1 Elementary Operators

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

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

definition list2policy :: ('a ⇒ 'b) list ⇒ ('a ⇒ 'b)
where
list2policy l = foldr (λ x y. (x ⊕ y)) l ∅

Determine the position of element of a list.

fun position :: 'a ⇒ 'a list ⇒ nat
where
position a [] = 0
| (position a (x#xs)) = (if a = x then 1 else (Suc (position a xs)))

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

fun applied-rule where
applied-rule C a (x#xs) = (if a ∈ dom (C x) then (Some x)
else (applied-rule C a xs))
| applied-rule C a [] = None

The following is used if the list is constructed backwards.

definition applied-rule-rev where
applied-rule-rev C a x = applied-rule C a (rev x)
The following is a typical policy transformation. It can be applied to any type of policy and removes all the rules from a policy with an empty domain. It takes two arguments: a semantic interpretation function and a list of rules.

fun rm-MT-rules where
rm-MT-rules C (x#xs) = (if dom (C x)= {} then rm-MT-rules C xs else x#(rm-MT-rules C xs))

|rm-MT-rules C [] = []

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

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

none-MT-rules C [] = True

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

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

|not-MT C [] = False

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

⟨proof⟩

lemma rmnMT: none-MT-rules C (rm-MT-rules C p)
⟨proof⟩

lemma rmnMT2: none-MT-rules C p ⇒ (rm-MT-rules C p) = p
⟨proof⟩

lemma nMTcharn: none-MT-rules C p = (∀ r ∈ set p. dom (C r) ≠ {}))
⟨proof⟩

lemma nMTeqSet: set p = set s ⇒ none-MT-rules C p = none-MT-rules C s
⟨proof⟩

lemma notMTnMT: [a ∈ set p; none-MT-rules C p] ⇒ dom (C a) ≠ {}
⟨proof⟩

lemma none-MT-rulesconc: none-MT-rules C (a@[b]) ⇒ none-MT-rules C a
⟨proof⟩

lemma nMTtail: none-MT-rules C p ⇒ none-MT-rules C (tl p)
lemma not-MTimpnotMT[simp]: not-MT C p \implies p \neq []

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

lemma NMPcharn: \[\{a \in \text{set } p; \text{dom } (C a) \neq \{\}\}\] \implies not-MT C p

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

Next, a few theorems about \textit{applied\_rule}:

lemma mrconc: \textit{applied\_rule\_rev} C x p = Some a \implies \textit{applied\_rule\_rev} C x (b\#p) = Some a

lemma mreq-end: \[\textit{applied\_rule\_rev} C x b = Some r; \textit{applied\_rule\_rev} C x c = Some r\]
\implies \textit{applied\_rule\_rev} C x (a\#b) = \textit{applied\_rule\_rev} C x (a\#c)

lemma mrconNone: \textit{applied\_rule\_rev} C x p = None \implies 
\textit{applied\_rule\_rev} C x (b\#p) = \textit{applied\_rule\_rev} C x [b]

lemma mreq-endNone: \[\textit{applied\_rule\_rev} C x b = None; \textit{applied\_rule\_rev} C x c = None\]
\implies \textit{applied\_rule\_rev} C x (a\#b) = \textit{applied\_rule\_rev} C x (a\#c)

lemma mreq-end2: \textit{applied\_rule\_rev} C x b = \textit{applied\_rule\_rev} C x c \implies 
\textit{applied\_rule\_rev} C x (a\#b) = \textit{applied\_rule\_rev} C x (a\#c)

lemma mreq-end3: \textit{applied\_rule\_rev} C x p \neq None \implies 
\textit{applied\_rule\_rev} C x (b \# p) = \textit{applied\_rule\_rev} C x (p)

lemma mrNoneMT: \[r \in \text{set } p; \textit{applied\_rule\_rev} C x p = None\] \implies 
x \notin \text{dom } (C r)
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. $P = P_1 P_2$) (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.

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

prod-list x [] par-comb ran-adapt dom-adapt = []
```

An instance, as usual there are four of them.

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

```haskell
lemma list2listNMT: x ≠ [] ⇒ map sem x ≠ []
```

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

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

```haskell
definition is-distr where
is-distr p = (λ g f. ( ∀ N P1 P2. ((g o-f ((p N (P1 ⊕ P2)) o f)) = ((g o-f ((p N P1) o f)) ⊕ (g o-f ((p N P2) o f)))))))
```

```haskell
definition is-strict where
is-strict p = (λ r d. ( ∀ P1. (r o-f (p P1 ∅ o d))) = ∅)
```

32
\textbf{lemma is-distr-orD}: is-distr \((\bigotimes \bigvee D)\) \(d\) \(r\)
\(\langle\text{proof}\rangle\)

\textbf{lemma is-strict-orD}: is-strict \((\bigotimes \bigvee D)\) \(d\) \(r\)
\(\langle\text{proof}\rangle\)

\textbf{lemma is-distr-2}: is-distr \((\bigotimes 2)\) \(d\) \(r\)
\(\langle\text{proof}\rangle\)

\textbf{lemma is-strict-2}: is-strict \((\bigotimes 2)\) \(d\) \(r\)
\(\langle\text{proof}\rangle\)

\textbf{lemma domStart}: \(t \in \text{dom } p1 \implies (p1 \bigoplus p2)\ t = p1\ t\)
\(\langle\text{proof}\rangle\)

\textbf{lemma notDom}: \(x \in \text{dom } A \implies \neg A\ x = \text{None}\)
\(\langle\text{proof}\rangle\)

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

\textbf{lemma Norm-Distr-1}: \((r \ o-f (((\bigotimes 1)\ P1 \ (\text{list2policy } P2))\ o\ d))\ x = (\text{list2policy } ((P1 \bigotimes_L P2) \ (\bigotimes 1)\ r\ d))\ x))\)
\(\langle\text{proof}\rangle\)

\textbf{lemma Norm-Distr-2}: \((r \ o-f (((\bigotimes 2)\ P1 \ (\text{list2policy } P2))\ o\ d))\ x = (\text{list2policy } ((P1 \bigotimes_L P2) \ (\bigotimes 2)\ r\ d))\ x))\)
\(\langle\text{proof}\rangle\)

\textbf{lemma Norm-Distr-A}: \((r \ o-f (((\bigotimes \bigvee A)\ P1 \ (\text{list2policy } P2))\ o\ d))\ x = (\text{list2policy } ((P1 \bigotimes_L P2) \ (\bigotimes \bigvee A)\ r\ d))\ x))\)
\(\langle\text{proof}\rangle\)

\textbf{lemma Norm-Distr-D}: \((r \ o-f (((\bigotimes \bigvee D)\ P1 \ (\text{list2policy } P2))\ o\ d))\ x = (\text{list2policy } ((P1 \bigotimes_L P2) \ (\bigotimes \bigvee D)\ r\ d))\ x))\)
\(\langle\text{proof}\rangle\)

Some domain reasoning

\textbf{lemma domSubsetDistr1}: \(\text{dom } A = \text{UNIV} \implies \text{dom } ((\lambda(x, y). \ x) \ o-f (A \bigotimes 1 B) \ o (\lambda x. \ (x,x))) = \text{dom } B\)
\(\langle\text{proof}\rangle\)

\textbf{lemma domSubsetDistr2}: \(\text{dom } A = \text{UNIV} \implies \text{dom } ((\lambda(x, y). \ x) \ o-f (A \bigotimes 2 B) \ o (\lambda x. \ (x,x))) = \text{dom } B\)
\(\langle\text{proof}\rangle\)
lemma domSubsetDistrA: dom A = UNIV ⇒ dom (((λ(x, y). x) o-f (A ⊗_A B)) o (λ x. (x,x))) = dom B
⟨proof⟩
lemma domSubsetDistrD: dom A = UNIV ⇒ dom (((λ(x, y). x) o-f (A ⊗_D B)) o (λ x. (x,x))) = dom B
⟨proof⟩
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))
⟨proof⟩

lemma PUTList-None: x /∈ dom (list2policy list) ⇒ PUTList PUT x list
⟨proof⟩

lemma PUTList-DomMT:
  (∀ y∈set list. dom a ∩ dom y = {}) ⇒ x ∈ (dom a) ⇒ x /∈ dom (list2policy list)
lemma distrPUTL2:
\[ x \in \text{dom } P \implies (\text{list2policy } PL) \ x = P \ x \implies \text{disjDom } PL \implies (\text{PUT } x = P \ x) \implies \text{PUTList } \text{PUT } x \ PL \]
⟨proof⟩

lemma distrPUTL:
\[ [x \in \text{dom } P; (\text{list2policy } PL) \ x = P \ x; \text{disjDom } PL] \implies (\text{PUT } x = P \ x) = \text{PUTList } \text{PUT } x \ PL \]
⟨proof⟩

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 \cup (gatherDomain-aux xs))
gatherDomain-aux [] = {}

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

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

definition disjDomGD where disjDomGD p = disjDom (butlast p)

lemma distrPUTLG1: [x \in \text{dom } P; (\text{list2policy } PL) \ x = P \ x; \text{PUTListGD } \text{PUT } x \ PL] \implies \text{PUT } x = P \ x
⟨proof⟩

lemma distrPUTLG2:
\[ PL \neq [] \implies x \in \text{dom } P \implies (\text{list2policy } (PL)) \ x = P \ x \implies \text{disjDomGD } PL \implies (\text{PUT } x = P \ x) \implies \text{PUTListGD } \text{PUT } x (PL) \]
⟨proof⟩

lemma distrPUTLG:
\[ [x \in \text{dom } P; (\text{list2policy } PL) \ x = P \ x; \text{disjDomGD } PL; PL \neq []] \implies (\text{PUT } x = P \ x) = \text{PUTListGD } \text{PUT } x \ PL \]
⟨proof⟩

end

35
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:

lemma 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.


\textbf{type-synonym} \( lr-id = \text{int} \)

\textbf{type-synonym} \( LR = lr-id \rightarrow (\text{user set}) \)

The security context stores all the existing LRs.

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

The user context stores the roles the users are in.

\textbf{type-synonym} \( \upsilon = \text{user} \rightarrow \text{role} \)

\subsection{3.1.2 Modelling Health Records and the Web Service API}

\textbf{Health Records}

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

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

\textbf{datatype} \( \text{status} = \text{Open} \mid \text{Closed} \)

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

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

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

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

\textbf{The Web Service API}

The operations provided by the service:

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

\textbf{fun} \( \text{is-createSCR where} \)
\textbf{fun} \( \text{is-appendEntry where} \)
\textbf{fun} \( \text{is-deleteEntry where} \)

38
\[ is\text{-}deleteEntry\ (deleteEntry\ u\ r\ p\ e\text{-}id) = True \]
| \[ is\text{-}deleteEntry\ x = False \]

**fun** is-readEntry **where**

\[ is\text{-}readEntry\ (readEntry\ u\ r\ p\ e) = True \]
| \[ is\text{-}readEntry\ x = False \]

**fun** is-readSCR **where**

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

**fun** is-changeStatus **where**

\[ is\text{-}changeStatus\ (changeStatus\ u\ r\ p\ s\ ei) = True \]
| \[ is\text{-}changeStatus\ x = False \]

**fun** is-deleteSCR **where**

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

**fun** is-addLR **where**

\[ is\text{-}addLR\ (addLR\ u\ r\ lrid\ lr\ us) = True \]
| \[ is\text{-}addLR\ x = False \]

**fun** is-removeLR **where**

\[ is\text{-}removeLR\ (removeLR\ u\ r\ p\ lr) = True \]
| \[ is\text{-}removeLR\ x = False \]

**fun** is-editEntry **where**

\[ is\text{-}editEntry\ (editEntry\ u\ r\ p\ e\text{-}id\ s) = True \]
| \[ is\text{-}editEntry\ x = False \]

**fun** SCROp \::\ (Operation \times DB) \rightarrow SCR **where**

\[ SCROp\ ((createSCR\ u\ r\ p),\ S) = S\ p \]
| \[ SCROp\ ((appendEntry\ u\ r\ p\ ei\ e),\ S) = S\ p \]
| \[ SCROp\ ((deleteEntry\ u\ r\ p\ e\text{-}id),\ S) = S\ p \]
| \[ SCROp\ ((readEntry\ u\ r\ p\ e),\ S) = S\ p \]
| \[ SCROp\ ((readSCR\ u\ r\ p),\ S) = S\ p \]
| \[ SCROp\ ((changeStatus\ u\ r\ p\ s\ ei),S) = S\ p \]
| \[ SCROp\ ((deleteSCR\ u\ r\ p),S) = S\ p \]
| \[ SCROp\ ((editEntry\ u\ r\ p\ e\text{-}id\ s),S) = S\ p \]
| \[ SCROp\ x = \bot \]

**fun** patientOfOp \::\ Operation \Rightarrow patient **where**

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

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

fun contentOfOp :: Operation ⇒ data where
  contentOfOp (appendEntry u r p e ei) = (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
allContentStatic (x#xs) = (contentStatic x \& allContentStatic xs)
|allContentStatic [] = True

3.1.3 Modelling Access Control

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

Sealed Envelopes

**type-synonym** SEPolicy = (Operation \times DB \rightarrow unit)

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

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

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

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

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

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

Legitimate Relationships

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

fun \textit{hasLR} :: user \Rightarrow patient \Rightarrow \Sigma \Rightarrow bool where
\textit{hasLR} u p \Sigma = (\textit{case} \Sigma p \textit{of} \ perpendicular \Rightarrow \textit{False}
\mid \lfloor \textit{lrs} \rfloor \Rightarrow (\exists \textit{lr}. \textit{lr} \in (\textit{ran lrs}) \land u \in \textit{lr}))

definition \textit{LRPolicy} :: \textit{LRPolicy} where
\textit{LRPolicy} = (\lambda (x, y). (\textit{if} \textit{hasLR} (\textit{userOfOp} x) (\textit{patientOfOp} x) \ y
\textit{then} \lfloor \textit{allow} () \rfloor
\textit{else} \lfloor \textit{deny} () \rfloor))

definition \textit{createSCRPolicy} :: \textit{LRPolicy} where
\textit{createSCRPolicy} x = (\textit{if} (\textit{is-createSCR} (\textit{fst} x))
\textit{then} \lfloor \textit{allow} () \rfloor
\textit{else} \perpendicular)

definition \textit{addLRPolicy} :: \textit{LRPolicy} where
\textit{addLRPolicy} x = (\textit{if} (\textit{is-addLR} (\textit{fst} x))
\textit{then} \lfloor \textit{allow} () \rfloor
\textit{else} \perpendicular)

definition \textit{LR-Policy} where \textit{LR-Policy} = \textit{createSCRPolicy} \oplus \textit{addLRPolicy} \oplus \textit{LR-Policy} \oplus \textit{A U}

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

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

fun \textit{createFunPolicy} :: \textit{FunPolicy} where
\textit{createFunPolicy} ((\textit{createSCR} u \ r \ p),(D,S)) = (\textit{if} p \in \textit{dom} D
\textit{then} \lfloor \textit{deny} () \rfloor
\textit{else} \lfloor \textit{allow} () \rfloor)
\mid \textit{createFunPolicy} x = \perpendicular
fun addLRFunPolicy :: FunPolicy where
addLRFunPolicy ((addLR u r p l us),(D,S)) = (if l ∈ dom S
then ⌊deny ()⌋
else ⌊allow ()⌋)

|addLRFunPolicy x = ⊥

fun removeLRFunPolicy :: FunPolicy where
removeLRFunPolicy ((removeLR u r p l),(D,S)) = (if l ∈ dom S
then ⌊allow ()⌋
else ⌊deny ()⌋)

|removeLRFunPolicy x = ⊥

fun readSCRFunPolicy :: FunPolicy where
readSCRFunPolicy ((readSCR u r p),(D,S)) = (if p ∈ dom D
then ⌊allow ()⌋
else ⌊deny ()⌋)

|readSCRFunPolicy x = ⊥

fun deleteSCRFunPolicy :: FunPolicy where
deleteSCRFunPolicy ((deleteSCR u r p),(D,S)) = (if p ∈ dom D
then ⌊allow ()⌋
else ⌊deny ()⌋)

|deleteSCRFunPolicy x = ⊥

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

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

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

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

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

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

Modelling Core RBAC

type-synonym RBACPolicy = Operation × v ⇒ unit

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

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

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 \( \bot \Rightarrow [S(p\rightarrow\emptyset)] \)
  | \( [x] \Rightarrow [S] \))
  | OpSuccessDB ((appendEntry u r p ei e),S) =
  (case S p of \( \bot \Rightarrow [S] \)
  | \( [x] \Rightarrow (\text{if } ei \in (\text{dom } x)
  \text{then } [S] 
  \text{else } [S(p \mapsto x(ei\rightarrow e))] ) \))
  | OpSuccessDB ((deleteSCR u r p),S) = (Some (S(p:=\bot)))
  | OpSuccessDB ((deleteEntry u r p ei),S) =
  (case S p of \( \bot \Rightarrow [S] \)
  | \( [x] \Rightarrow \text{Some } (S(p \mapsto \text{Map.empty lv})) \))
  | \( \bot \Rightarrow [S] \))
  | OpSuccessDB ((changeStatus u r p ei s),S) =
  (case S p of \( \bot \Rightarrow [S] \)
  | \( [x] \Rightarrow (\text{case } x ei of }
  \lfloor e \rfloor \Rightarrow [S(p \mapsto x(ei\rightarrow(s,snd e)))] 
  \lfloor \bot \rfloor \Rightarrow [S]) \))
  | OpSuccessDB ((editEntry u r p ei e),S) =
  (case S p of \( \bot \Rightarrow [S] \)
  | \( [x] \Rightarrow (\text{case } x ei of }
  \lfloor e \rfloor \Rightarrow [S(p \mapsto x(ei\rightarrow e))]
  \lfloor \bot \rfloor \Rightarrow [S]) \))
  | OpSuccessDB ((x,S) = [S])

fun OpSuccessSigma :: (Operation \times \Sigma) \rightarrow \Sigma where
  OpSuccessSigma (addLR u r p lv-id us,S) =
  (case S p of \[lv\] \Rightarrow (case \( \text{lv} \text{lv}-id \) of }
  \( \bot \Rightarrow [S(p\rightarrow((\text{lv}\text{lv}-id\rightarrow us)))] )
  | \( [x] \Rightarrow [S] ) \))
  | \( \bot \Rightarrow [S(p\rightarrow(\text{Map.empty(lv-id\rightarrow us))})] )
  | OpSuccessSigma (removeLR u r p lv-id,S) =

45
(case \( S \ p \) of Some \( lrs \) ⇒ \([S(p-->\text{lr}(\text{lr-id}:=\bot))]\))
| \( \bot \) ⇒ \([S]\))
\(\text{OpSuccessSigma} \ (x,S) = [S]\)

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

Output

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

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

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

3.1.5 Combine All Parts

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

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

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_\( f (\text{RBACPolicy} \bigotimes \nu \text{SE-LR-FUN-Policy}) \)
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
\( \text{ST-Allow} = ((\text{OpSuccessOutput} \bigotimes \_ \text{M} (\text{OpSuccessDB} \bigotimes \_ \nu \text{OpSuccessSigma} \bigotimes \_ \text{OpSuccessUC}))) \)
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
\( \text{ST-Deny} = (\lambda (\text{ope,sp,si,uc}). \text{Some} ((\text{}), \text{sp,si,uc})) \)

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

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

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

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

end

### 3.2 Instantiating Our Secure Service Example

theory

\( ServiceExample \)

imports

\( Service \)

begin

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

#### 3.2.1 Access Control Configuration

**definition** \( alice :: user \) where \( alice = 1 \)

**definition** \( bob :: user \) where \( bob = 2 \)

**definition** \( charlie :: user \) where \( charlie = 3 \)

**definition** \( patient1 :: patient \) where \( patient1 = 5 \)

**definition** \( patient2 :: patient \) where \( patient2 = 6 \)

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

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

**definition** \( entry1 :: entry \) where

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

**definition** \( entry2 :: entry \) where

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

**definition** \( entry3 :: entry \) where

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

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

\( SCR1 = (\text{Map.empty}(1\rightarrow entry1)) \)
definition \( SCR_2 :: SCR \) where
\[ SCR_2 = (\text{Map.empty}) \]

definition \( Spine_0 :: DB \) where
\[ Spine_0 = \text{Map.empty}(\text{patient1} \mapsto SCR_1)(\text{patient2} \mapsto SCR_2) \]

definition \( LR_1 :: LR \) where
\[ LR_1 = (\text{Map.empty}(1 \mapsto \{\text{alice}\})) \]

definition \( \Sigma_0 :: \Sigma \) where
\[ \Sigma_0 = (\text{Map.empty}(\text{patient1} \mapsto LR_1)) \]

3.2.2 The Initial System State
definition \( \sigma_0 :: DB \times \Sigma \times u \) where
\[ \sigma_0 = (Spine_0, \Sigma_0, \text{UC}_0) \]

3.2.3 Basic Properties

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

\langle proof \rangle

**lemma \([\text{cong,simp}]\):**
\[ ((\text{if hasLR urp1-alice 1 } \Sigma_0 \text{ then } \text{allow () } \text{ else } \text{deny ()}) = \bot) = False \]

\langle proof \rangle

**lemmas** \( \text{MonSimps} = \text{valid-SE-def unit-SE-def bind-SE-def} \)

**lemmas** \( \text{Psplits} = \text{option.splits unit.splits prod.splits decision.splits} \)

**lemmas** \( \text{PolSimps} = \text{valid-SE-def unit-SE-def bind-SE-def if-splits policy2MON-def} \)

\[ \text{SE-LR-RBAC-ST-Policy-def map-add-def id-def LRsimps prod-2-def} \]
\[ \text{RBACPolicy-def} \]
\[ \text{SE-LR-Policy-def SEPolicy-def RBAC-def deleteEntrySE-def editEntrySE-def} \]
\[ \text{readEntrySE-def } \sigma_0-def \Sigma_0-def \text{UC}_0-def \text{patient1-def patient2-def LR1-def} \]
\[ \text{alice-def bob-def charlie-def get-entry-def SE-LR-RBAC-Policy-def Allow-def} \]
\[ \text{Deny-def dom-restrict-def policy-range-comp-def prod-orA-def prod-orD-def} \]
\[ \text{ST-Allow-def ST-Deny-def Spine0-def SCR1-def SCR2-def entry1-def} \]
\[ \text{entry2-def} \]
\[ \text{entry3-def FunPolicy-def SE-LR-FUN-Policy-def o-def image-def UPFDefs} \]
lemma SE-LR-RBAC-Policy $(\langle \text{createSCR alice Clerical patient1} \rangle, \sigma_0) = \text{Some } (\text{deny }())$
\begin{equation*}
\langle \text{proof} \rangle
\end{equation*}

lemma exBool[simp]: $\exists a :: \text{bool}. a$
\begin{equation*}
\langle \text{proof} \rangle
\end{equation*}

lemma deny-allow[simp]: $\lfloor \text{deny }() \rfloor \notin \text{Some } \text{'} \text{ range allow}$
\begin{equation*}
\langle \text{proof} \rangle
\end{equation*}

lemma allow-deny[simp]: $\lfloor \text{allow }() \rfloor \notin \text{Some } \text{'} \text{ range deny}$
\begin{equation*}
\langle \text{proof} \rangle
\end{equation*}

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

lemma
$$(\sigma_0 \models (os \leftarrow \text{mbind } [(\text{createSCR alice Clerical patient1})] \text{ (PolMon);}\n\quad \text{return } (os = [(\text{deny } (\text{Out} ))])))$$
\begin{equation*}
\langle \text{proof} \rangle
\end{equation*}

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

lemma SE-LR-RBAC-Policy $(\langle \text{appendEntry alice Clerical patient1 ci d} \rangle, \sigma_0) = [(\text{deny }())$
\begin{equation*}
\langle \text{proof} \rangle
\end{equation*}

end
4 Conclusion and Related Work

4.1 Related Work

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

4.2 Conclusion Future Work

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

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

5.1 Basic Monad Theory for Sequential Computations

theory
  Monads
imports
  Main
begin

5.1.1 General Framework for Monad-based Sequence-Test

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

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

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

State Exception Monads

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

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

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

notation bind-SE (bind_{SE})
syntax
translations
\[ x ← f; \ g \Rightarrow \ CONST \ bind-SE \ f \ (\% \ x . \ g) \]

**definition** unit-SE :: 'o ⇒ ('o, 'σ)MON_SE

**where** unit-SE e = (λσ. Some(e,σ))

**notation** unit-SE (unit_SE)

**definition** fail_SE :: ('o, 'σ)MON_SE

**where** fail_SE = (λσ. None)

**notation** fail_SE (fail_SE)

**definition** assert-SE :: ('σ ⇒ bool) ⇒ (bool, 'σ)MON_SE

**where** 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** assume-SE P = (λσ. if ∃σ . P σ then Some((), SOME σ . P σ) 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** if-SE e E F = (λσ. if e σ then E σ 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

⟨proof⟩

**lemma** bind-right-unit: (x ← m; return x) = m

⟨proof⟩

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

⟨proof⟩

In order to express test-sequences also on the object-level and to make our theory amenable to formal reasoning over test-sequences, we represent them as lists of input and generalize the bind-operator of the state-exception monad accordingly. The approach is straightforward, but comes with a price: we have to encapsulate all input and output data into one type. Assume that we have a typed interface to a module with the operations op_1, op_2, ..., op_n with the inputs i_1, i_2, ..., i_n (outputs are treated analogously). Then we can encode for this interface the general input - type:

```
datatype in = op_1 :: i_1 | ... | i_n
```

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

In order to express test-sequences also on the object-level and to make our theory amenable to formal reasoning over test-sequences, we represent them as lists of input and generalize the bind-operator of the state-exception monad accordingly. Thus, the notion of test-sequence is mapped to the notion of a \textit{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 loose some type-safety in this approach; we have to express that in traces only corresponding input and output belonging to the same operation will occur; this form of side-conditions have to be expressed inside HOL. From the user perspective, this will not make much difference, since junk-data resulting from too weak typing can be ruled out by adopted front-ends.

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

\begin{verbatim}
fun mbind :: 'i list ⇒ ('i ⇒ ('o,'σ) MON SE) ⇒ ('o list,'σ) MON SE
  where mbind [] iostep σ = Some([], σ) |
    mbind (a#H) iostep σ =
      (case iostep a σ of
        None ⇒ Some([], σ) |
        Some (out, σ') ⇒ (case mbind H iostep σ' of
          None ⇒ Some([out],σ') |
          Some( ots,σ'') ⇒ Some(out#ots,σ''))

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

lemma mbind-unit [simp]: mbind [] f = (return [])
  ⟨proof⟩

lemma mbind-nofailure [simp]: mbind S f σ ≠ None
\end{verbatim}
The fail-strict version of $\text{mbind}'$ looks as follows:

\[
\begin{aligned}
\text{fun} & \quad \text{mbind}' :: \text{'t list} \Rightarrow (\text{'t} \Rightarrow (\text{'o, 'σ}) \text{MON}_SE) \Rightarrow (\text{'o list, 'σ}) \text{MON}_{SE} \\
\text{where} & \quad \text{mbind}' [\text{} \text{iostep \ σ} = \text{Some([], \ σ)} \\
& \quad \text{mbind}' (a \# H) \text{iostep \ σ} = \\
& \quad \quad (\text{case \ iostep \ a \ σ \ of} \\
& \quad \quad \quad \text{None} \Rightarrow \text{None} \\
& \quad \quad \quad \text{Some (out, \ σ')} \Rightarrow (\text{case \ mbind \ H \ iostep \ σ'} \ of \\
& \quad \quad \quad \quad \text{None} \Rightarrow \text{None} \quad \text{— fail-strict} \\
& \quad \quad \quad \text{Some(outs, σ'')} \Rightarrow \text{Some(outs', σ'')})
\end{aligned}
\]

$\text{mbind'}$ as failure strict operator can be seen as a foldr on bind—if the types would match . . .

\[
\begin{aligned}
\text{definition} & \quad \text{try-SE :: ('}o, 'σ) \text{MON}_{SE} \Rightarrow ('}o \text{ option, 'σ) \text{MON}_{SE} \\
\text{where} & \quad \text{try-SE ioprog} = (\lambda \ σ. \ \text{case \ ioprog \ σ \ of} \\
& \quad \quad \text{None} \Rightarrow \text{Some(None, \ σ)} \\
& \quad \quad \text{Some(outs, σ')} \Rightarrow \text{Some(Some outs, σ')})
\end{aligned}
\]

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

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

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

\[
\begin{aligned}
\text{definition} & \quad \text{alt-SE :: [('}o, 'σ)\text{MON}_{SE}, ('}o, 'σ)\text{MON}_{SE}] \Rightarrow ('}o, 'σ)\text{MON}_{SE} \quad (\text{infixl} \\
& \quad \text{\texttt{⊓}} \text{SE} 10) \\
\text{where} & \quad (f \ \texttt{⊓}_SE \ g) = (\lambda \ σ. \ \text{case} \ f \ σ \ \text{of} \ \text{None} \Rightarrow g \ σ \\
& \quad \quad \text{Some \ H} \Rightarrow \text{Some \ H})
\end{aligned}
\]

\[
\begin{aligned}
\text{definition} & \quad \text{malt-SE :: ('}o, 'σ)\text{MON}_{SE} \text{list} \Rightarrow ('}o, 'σ)\text{MON}_{SE} \\
\text{where} & \quad \text{malt-SE} \ S = \text{foldr} \ \text{alt-SE} \ S \ \text{fail}_SE \\
\text{notation} & \quad \text{malt-SE} (\bigcap_{SE})
\end{aligned}
\]

\[
\begin{aligned}
\text{lemma} \ & \quad \text{malt-SE-mt} \ [\text{simp}]: \bigcap_{SE} [] = \text{fail}_SE
\end{aligned}
\]
lemma \textit{malt-SE-cons} [simp]: $\prod_{SE} (a \# S) = (a \cap_{SE} (\prod_{SE} S))$

\langle proof \rangle

\textbf{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.

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

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

\textbf{notation} \textit{bind-SB} \((\text{bind}_{SB})\)

definition \textit{unit-SB} :: \(\circ \Rightarrow (\circ', \sigma) \text{MON}_{SB}\)

\textbf{notation} \textit{unit-SB} \((\text{unit}_{SB})\)

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

\textbf{translations}

\begin{align*}
& x := f; g \Rightarrow \text{CONST bind-SB} f (\% x . g) \\
& \text{lemma \textit{bind-left-unit-SB}} : (x := \text{returns } a; m) = m \\
& \langle proof \rangle \\
& \text{lemma \textit{bind-right-unit-SB}} : (x := m; \text{returns } x) = m \\
& \langle proof \rangle \\
& \text{lemma \textit{bind-assoc-SB}} : (y := (x := m; k); h) = (x := m; (y := k; h)) \\
& \langle proof \rangle \\
\end{align*}

\textbf{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.

\textbf{type-synonym} \((\circ, \sigma) \text{MON}_{SBE} = \circ \Rightarrow ((\circ \times \sigma) \text{ set}) \text{ option}\)

definition \textit{bind-SBE} :: \((\circ, \sigma) \text{MON}_{SBE} \Rightarrow (\circ \Rightarrow (\circ', \sigma) \text{MON}_{SBE}) \Rightarrow (\circ', \sigma) \text{MON}_{SBE}\)
where \[ \text{bind-SBE } f = \lambda \sigma. \text{case } f \sigma \text{ of None } \Rightarrow \text{None} \]
\[ \text{Some } S \Rightarrow (\lambda \sigma. \text{out } \sigma') \cdot g \out \sigma' \text{ \' } S \]
\[ \text{if None } \in S' \text{ then None} \]
\[ \text{else Some}(\bigcup (\text{the } \text{'} S')) \]

syntax -bind-SBE :: [pttrn,('o,'σ)MON_SBE,(('o,'σ)MON_SBE)] ⇒ ('o,'σ)MON_SBE

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

definition unit-SBE :: ('o) ⇒ ('o,'σ)MON_SBE ((returning -) 8)
where unit-SBE e = (λσ. Some({(e,σ)}))

notation assert-SBE (assert_SBE)

definition assert-SBE :: ('σ ⇒ bool) ⇒ (unit, 'σ)MON_SBE
where assert-SBE e = (λσ. if e σ then Some({((),σ)})
\[ \text{else None} \]

notation assert-SBE (assert_SBE)

definition assume-SBE :: ('σ ⇒ bool) ⇒ (unit, 'σ)MON_SBE
where assume-SBE e = (λσ. if e σ then Some({((),σ)})
\[ \text{else Some } \{\} \]

notation assume-SBE (assume_SBE)

definition havoc-SBE :: (unit, 'σ)MON_SBE
where havoc-SBE = (λσ. Some({x. True}))

notation havoc-SBE (havoc_SBE)

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

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

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

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

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 :: 'σ ⇒ (bool,'σ) MON_SE ⇒ bool (infix \[\text{=} 15\])
where \[ \sigma \models m = (m \sigma \neq \text{None} \wedge \text{fst}(\text{the } (m \sigma))) \]
This notation considers failures as valid—a definition inspired by I/O conformance. Note that it is not possible to define this concept once and for all in a Hindley-Milner type-system. For the moment, we present it only for the state-exception monad, although for the same definition, this notion is applicable to other monads as well.

**Lemma syntax-test:**
\[ \sigma \vdash (os \leftarrow (mbind \ i s \ ioprog); \text{return}(length \ i s = length \ os)) \]

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

**Proof**
Recall mbind unit for the base case.

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

**Proof**

**Lemma valid-failure':**
\[ A \ \sigma = \text{None} \implies \neg (\sigma \vdash ((s \leftarrow A \ M s))) \]

**Proof**

**Lemma valid-successElem:**
\[ M \ \sigma = \text{Some}(f,\sigma) \implies (\sigma \vdash M) = f \ \sigma \]

**Proof**

**Lemma valid-success:**
\[ \text{ioprog} \ a \ \sigma = \text{Some}(b,\sigma') \implies (\sigma \vdash (s \leftarrow \text{mbind} (a\#S) \ ioprog \ M s)) = (\sigma' \vdash (s \leftarrow \text{mbind} S \ ioprog \ M (b\#s))) \]

**Proof**

**Lemma valid-success'':**
\[ \text{ioprog} \ a \ \sigma = \text{Some}(b,\sigma') \implies (\sigma \vdash (s \leftarrow \text{mbind} (a\#S) \ ioprog \ \text{return} (P s))) = (\sigma' \vdash (s \leftarrow \text{mbind} S \ ioprog \ \text{return} (P (b\#s)))) \]

**Proof**

**Lemma valid-success':**
\[ A \ \sigma = \text{Some}(b,\sigma') \implies (\sigma \vdash ((s \leftarrow A \ M s))) = (\sigma' \vdash (M b)) \]

**Proof**

**Lemma valid-both:**
\[ (\sigma \vdash (s \leftarrow \text{mbind} (a\#S) \ ioprog \ \text{return} (P s))) = \]
\[ \text{case} \ \text{ioprog} \ a \ \sigma \ \text{of} \]
\[ \text{None} \Rightarrow (\sigma \vdash (\text{return} (P []))) \]
\[ \text{Some}(b,\sigma') \Rightarrow (\sigma' \vdash (s \leftarrow \text{mbind} S \ ioprog \ \text{return} (P (b\#s)))) \]

59
\begin{proof}

\textbf{lemma valid-propagate-1 [simp]}: \((\sigma \models (\text{return } P)) = (P)\)
\end{proof}

\begin{proof}

\textbf{lemma valid-propagate-2}: \(\sigma \models ((s \leftarrow A ; M s)) \implies \exists v \sigma'. \text{the}(A \sigma) = (v,\sigma') \land \sigma'\)
\end{proof}

\begin{proof}

\textbf{lemma valid-propagate-2'}: \(\sigma \models ((s \leftarrow A ; M s)) \implies \exists v \sigma'. A \sigma = \text{Some } a \land (\text{snd } a)\)
\end{proof}

\begin{proof}

\textbf{lemma valid-propagate-2''}: \(\sigma \models ((s \leftarrow A ; M s)) \implies \exists v \sigma'. A \sigma = \text{Some } (v,\sigma') \land \sigma'\)
\end{proof}

\begin{proof}

\textbf{lemma valid-propoagate-3 [simp]}: \((\sigma_0 \models (\lambda \sigma. \text{Some } (f \sigma, \sigma))) = (f \sigma_0)\)
\end{proof}

\begin{proof}

\textbf{lemma valid-propoagate-3'}: \neg(\sigma_0 \models (\lambda \sigma. \text{None}))
\end{proof}

\begin{proof}

\textbf{lemma assert-disch1} : \(P \sigma \implies (\sigma \models (x \leftarrow \text{assert}_SE P; M x)) = (\sigma \models (M \text{True}))\)
\end{proof}

\begin{proof}

\textbf{lemma assert-disch2} : \neg P \sigma \implies \neg (\sigma \models (x \leftarrow \text{assert}_SE P; M s))
\end{proof}

\begin{proof}

\textbf{lemma assert-disch3} : \neg P \sigma \implies \neg (\sigma \models (\text{assert}_SE P))
\end{proof}

\begin{proof}

\textbf{lemma assert-D} : \((\sigma \models (x \leftarrow \text{assert}_SE P; M x)) \implies P \sigma \land (\sigma \models (M \text{True}))\)
\end{proof}

\begin{proof}

\textbf{lemma assume-D} : \((\sigma \models (x \leftarrow \text{assume}_SE P; M x)) \implies \exists \sigma. (P \sigma \land (\sigma \models (M ()))\)
\end{proof}

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).

\textbf{lemma if-SE-D1} : \(P \sigma \implies (\sigma \models \text{if}_SE P \ B_1 \ B_2) = (\sigma \models B_1)\)
\end{proof}
\textbf{lemma} if-SE-D2 : \( \neg P \sigma \implies (\sigma \models if_{SE} P B_1 B_2) = (\sigma \models B_2) \) \\
\hspace{1em} \langle \text{proof} \rangle \\

\textbf{lemma} if-SE-split-asn : \( (\sigma \models if_{SE} P B_1 B_2) = ((P \sigma \land (\sigma \models B_1)) \lor (\neg P \sigma \land (\sigma \models B_2))) \) \\
\hspace{1em} \langle \text{proof} \rangle \\

\textbf{lemma} if-SE-split : \( (\sigma \models if_{SE} P B_1 B_2) = ((P \sigma \rightarrow (\sigma \models B_1)) \land (\neg P \sigma \rightarrow (\sigma \models B_2))) \) \\
\hspace{1em} \langle \text{proof} \rangle \\

\textbf{lemma} \([\text{code}]\) : \( (\sigma \models m) = (\text{case } (m \sigma) \text{ of } \text{None } \Rightarrow \text{False } \mid (\text{Some } (x,y)) \Rightarrow x) \) \\
\hspace{1em} \langle \text{proof} \rangle \\

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.

\textbf{definition} valid-SBE :: \( 'a \Rightarrow (a,\sigma) \text{ MON}_{SBE} \Rightarrow \text{ bool } (\text{infix } \models_{SBE} 15) \) \\
\hspace{1em} \text{where } \sigma \models_{SBE} m \equiv (m \sigma \neq \text{None}) \\

This notation considers all non-failures as valid.

\textbf{lemma} assume-assert : \( (\sigma \models_{SBE} (\text{- } :\equiv \text{ assume}_{SBE} P ; \text{ assert}_{SBE} Q)) = (P \sigma \rightarrow Q \sigma) \) \\
\hspace{1em} \langle \text{proof} \rangle \\

\textbf{lemma} assert-intro : \( Q \sigma \implies \sigma \models_{SBE} (\text{ assert}_{SBE} Q) \) \\
\hspace{1em} \langle \text{proof} \rangle \\

end
Bibliography


