The Unified Policy Framework (UPF)

Achim D. Brucker* Lukas Brügger† Burkhart Wolff‡

*Achim.brucker@sap.com
†Lukas.A.Bruegger@gmail.com
‡burkhart.wolff@lri.fr

September 19, 2015

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 19
      2.3.2 Policy Composition 20
   2.4 Parallel Composition 22
      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 28
      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
1 Introduction

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

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

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

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

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

2.1 The Core of the Unified Policy Framework (UPF)

theory

UPFCore

imports

Monads

begin

2.1.1 Foundation

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

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

datatype $\alpha$ decision = allow $\alpha$ | deny $\alpha$

type-synonym ($\alpha, \beta$) policy = $\alpha \rightarrow \beta$ decision (infixr $\mid\rightarrow\rangle$)

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

translations (type) $\alpha \mid\rightarrow\rangle \beta$ <= (type) $\alpha \rightarrow \beta$ decision

type-notation (xsymbols) policy (infixr $\leftrightarrow\rangle$)
... allowing the notation \( \alpha \mapsto \beta \) for the policy type and the alternative notations for None and Some of the HOL library \( \alpha \) option type:

**notation** None (⊥)

**notation** Some ([·] 80)

Thus, the range of a policy may consist of \( \lceil \text{accept } x \rceil \) data, of \( \lceil \text{deny } x \rceil \) 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** Allow :: \( \alpha \) decision set

**where** Allow = range allow

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

**where** Deny = range deny

### 2.1.2 Policy Constructors

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

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

Furthermore, we add notation adopted to our problem domain:

**nonterminal** policylets and policylet

**syntax**

\[
\begin{align*}
policylet1 &:: [\alpha, \alpha] => policylet \quad (-/+=/-) \\
policylet2 &:: [\alpha, \alpha] => policylet \quad (-/-/=/-) \\
\quad &:: policylet => policylets \quad (-) \\
\text{Maplets} &:: [\text{policylet, policylets}] => policylets (-/, -) \\
\text{Maplets} &:: [\text{policylet, policylets}] => policylets (-/, -) \\
\text{MapUpd} &:: [\alpha \mapsto \beta, \text{policylets}] => 'a => 'b \quad (-/\{\cdot\}[900,0]900)
\end{align*}
\]

**syntax (xsymbols)**

\[
\begin{align*}
policylet1 &:: \langle'\alpha', '\alpha'\rangle => policylet \quad (-/\{\cdot\}/) \\
policylet2 &:: \langle'\alpha', '\alpha'\rangle => policylet \quad (-/\{\cdot\}/) \\
\text{emptypolicy} &:: 'a \mapsto 'b \quad (\emptyset)
\end{align*}
\]

**translations**

\[
\begin{align*}
\text{MapUpd } m \text{ (-Maplets } xy \text{ ms}) &\Rightarrow \text{MapUpd } (-\text{MapUpd } m \text{ xy }) \text{ ms} \\
\text{MapUpd } m \text{ (-policylet1 } x \text{ y) } &\Rightarrow m(x := \text{CONST Some (CONST allow y)}) \\
\text{MapUpd } m \text{ (-policylet2 } x \text{ y) } &\Rightarrow m(x := \text{CONST Some (CONST deny y)})
\end{align*}
\]
Here are some lemmas essentially showing syntactic equivalences:

**lemma test**: $\emptyset (x \mapsto + a, y \mapsto - b) = \emptyset (x \mapsto + a, y \mapsto - b)$  ⟨proof⟩

**lemma test2**: $p(x\mapsto + a, x\mapsto - b) = p(x\mapsto - b)$  ⟨proof⟩

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

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

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

**lemma pol-upd-allow-nonempty**: $t (k \mapsto + x) \not= \emptyset$

**lemma pol-upd-deny-nonempty**: $t (k \mapsto - x) \not= \emptyset$

**lemma pol-upd-eqD1**: $m (a \mapsto + x) = n (a \mapsto + y) \implies x = y$

**lemma pol-upd-eqD2**: $m (a \mapsto - x) = n (a \mapsto - y) \implies x = y$

**lemma pol-upd-neq1** [simp]: $m (a \mapsto + x) \not= n (a \mapsto - y)$

### 2.1.3 Override Operators

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

**syntax**

```
-policyoverride :: \[a \mapsto \text{'b}, \text{'a} \mapsto \text{'b}\] \mapsto \text{'a} \mapsto \text{'b} (\text{infixl } (+/)) 100)
```

**syntax (xsymbols)**

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

**translations**

```
p \oplus q \equiv q ++ p
```
Some elementary facts inherited from Map are:

**lemma** override-empty: \( p \bigoplus \emptyset = p \)

\( \langle \text{proof} \rangle \)

**lemma** empty-override: \( \emptyset \bigoplus p = p \)

\( \langle \text{proof} \rangle \)

**lemma** override-assoc: \( p_1 \bigoplus (p_2 \bigoplus p_3) = (p_1 \bigoplus p_2) \bigoplus p_3 \)

\( \langle \text{proof} \rangle \)

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

**definition** override-A :: \([\alpha \mapsto \beta, \alpha \mapsto \beta] \Rightarrow [\alpha \mapsto \beta]\) (infixl \(+ + \cdot A\ 100\))

where \( m_2 ++ - A m_1 = \)

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

**syntax** (xsymbols)

- policyoverride-A :: \('[a \mapsto 'b, 'a \mapsto 'b] \Rightarrow 'a \mapsto 'b\) (infixl \(\bigoplus\cdot A\ 100\))

**translations**

\( p \bigoplus A q \equiv p ++ \cdot A q \)

**lemma** override-A-empty[simp]: \( p \bigoplus A \emptyset = p \)

\( \langle \text{proof} \rangle \)

**lemma** empty-override-A[simp]: \( \emptyset \bigoplus A p = p \)

\( \langle \text{proof} \rangle \)

**lemma** override-A-assoc: \( p_1 \bigoplus A (p_2 \bigoplus A p_3) = (p_1 \bigoplus A p_2) \bigoplus A p_3 \)

\( \langle \text{proof} \rangle \)

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

where \( m_1 ++ \cdot D m_2 = \)

\((\lambda x. (\text{case } m_2 x \text{ of } [\text{deny } a] \Rightarrow [\text{deny } a]
| [\text{allow } a] \Rightarrow (\text{case } m_1 x \text{ of } [\text{deny } b] \Rightarrow [\text{deny } b]
| - \Rightarrow [\text{allow } a])
| \bot \Rightarrow m_1 x)) \)
syntax (xsymbols)
-policyoverride-D :: ['a → 'b, 'a → 'b] ⇒ 'a → 'b (infixl ⊕_D 100)
translations
p ⊕_D q ⇔ p ++_D q

lemma override-D-empty[simp]: p ⊕_D ∅ = p
⟨proof⟩

lemma empty-override-D[simp]: ∅ ⊕_D p = p
⟨proof⟩

lemma override-D-assoc: p1 ⊕_D (p2 ⊕_D p3) = (p1 ⊕_D p2) ⊕_D p3
⟨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 :: ['β⇒'γ, 'α→'β] ⇒ 'α→'γ (infixl o′-f 55)
where
f o′-f p = (λx. case p x of
    ⌊allow y⌋ ⇒ ⌊allow (f y)⌋
  | ⌊deny y⌋ ⇒ ⌊deny (f y)⌋
  | ⊥ ⇒ ⊥)

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

lemma policy-range-comp-strict : f o′-f ∅ = ∅
⟨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 :: [('β⇒'γ)×('β⇒'γ),'α→'β] ⇒ 'α→'γ (infixr ∇ 100)
where (P) ∇ p = (λx. case p x of
    ⌊allow y⌋ ⇒ ⌊allow ((fst P) y)⌋
\begin{align*}
&\left[ \text{deny } y \right] \Rightarrow \left[ \text{deny } (\text{snd } P \ y) \right] \\
&\perp \Rightarrow \perp
\end{align*}

\textbf{lemma} range-split-strict[simp]: $P \nrightarrow \emptyset = \emptyset$

\text{⟨proof⟩}

\textbf{lemma} range-split-charn:
\begin{align*}
(f, g) \nrightarrow p &= (\lambda x. \text{case } p x \text{ of} \\
&\quad \left[ \text{allow } x \right] \Rightarrow \left[ \text{allow } (f \ x) \right] \\
&\quad \left[ \text{deny } x \right] \Rightarrow \left[ \text{deny } (g \ x) \right] \\
&\quad \perp \Rightarrow \perp
\end{align*}

\text{⟨proof⟩}

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

\textbf{lemma} range-split-vs-range-compose: $(f, f) \nrightarrow p = f \circ f \ p$

\text{⟨proof⟩}

\textbf{lemma} range-split-id [simp]: $(\text{id}, \text{id}) \nrightarrow p = p$

\text{⟨proof⟩}

\textbf{lemma} range-split-bi-compose [simp]: $(f_1, f_2) \nrightarrow (g_1, g_2) \nrightarrow p = (f_1 \circ g_1, f_2 \circ g_2) \nrightarrow p$

\text{⟨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.

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

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

\textbf{definition} range-split2 :: $[(\alpha \Rightarrow \gamma) \times (\alpha \Rightarrow \gamma), \alpha \mapsto \beta] \Rightarrow (\alpha \Rightarrow (\beta \times \gamma)) \quad \text{(infixr } \nrightarrow 2 100)$
where \( P \nabla 2 p = (\lambda x. \text{case } p x \text{ of}
\begin{align*}
\lambda & \Rightarrow \lambda (y, (\text{fst } P) x) \\
\text{deny } y & \Rightarrow \text{deny } (y, (\text{snd } P) x) \\
\bot & \Rightarrow \bot
\end{align*}
\)

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

where\(\text{policy2MON} p = (\lambda i . \text{case } p (i, \sigma) \text{ of}
\begin{align*}
\llbracket \text{allow } (\text{outs}, \sigma') \rrbracket & \Rightarrow \llbracket \text{allow } \text{outs}, \sigma' \rrbracket \\
\llbracket \text{deny } (\text{outs}, \sigma') \rrbracket & \Rightarrow \llbracket \text{deny } \text{outs}, \sigma' \rrbracket \\
\bot & \Rightarrow \bot
\end{align*}
\)

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

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 build by composing the elementary policies defined in this theory.

2.2.1 The Core Policy Combinators: Allow and Deny Everything

definition\(\text{deny-pfun} :: ('\alpha \mapsto '\beta) \Rightarrow ('\alpha \mapsto '\beta) (\text{AllD})\)

where\(\text{deny-pfun} pf \equiv (\lambda x . \text{case } pf x \text{ of}
\begin{align*}
y & \Rightarrow [\text{deny } (y)] \\
\bot & \Rightarrow \bot
\end{align*}
\)

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

where
allow-pfun pf \equiv (\lambda x. \text{case pf x of} \left[ \begin{array}{l} y \Rightarrow [\text{allow } (y)] \\ \bot \Rightarrow \bot \end{array} \right])

**syntax** (xsymbols)
-allow-pfun :: ('\alpha \rightarrow '\beta) \Rightarrow ('\alpha \mapsto '\beta) (A_p)

**translations**
\[ A_p \ f = \textbf{AllA} \ f \]

**syntax** (xsymbols)
-deny-pfun :: ('\alpha \rightarrow '\beta) \Rightarrow ('\alpha \mapsto '\beta) (D_p)

**translations**
\[ D_p \ f = \textbf{AllD} \ f \]

**notation** (xsymbols)
\begin{align*}
deny-pfun \ (\textbf{binder } \forall D \ 10) \text{ and } \\
allow-pfun \ (\textbf{binder } \forall A \ 10)
\end{align*}

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

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

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

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

**lemma** AllA-apply[simp]: (\forall Ax. Some (P x)) x = [allow (P x)]
\langle \text{proof} \rangle

**lemma** AllD-apply[simp]: (\forall Dx. Some (P x)) x = [deny (P x)]
\langle \text{proof} \rangle

**lemma** neq-Allow-Deny: pf \neq \emptyset \implies (deny-pfun pf) \neq (allow-pfun pf)
\langle \text{proof} \rangle

2.2.2 Common Instances

**definition** allow-all-fun :: ('\alpha \Rightarrow '\beta) \Rightarrow ('\alpha \mapsto '\beta) (A_f)
\begin{align*}
\text{where allow-all-fun } f &= \text{ allow-pfun (Some } o \ f) \end{align*}
**definition** deny-all-fun :: (\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \beta) (D_f)
where deny-all-fun f \equiv deny-pfun (Some \circ f)

**definition**
deny-all-id :: \alpha \Rightarrow \alpha (D_I) where
deny-all-id \equiv deny-pfun (id \circ Some)

**definition**
allow-all-id :: \alpha \Rightarrow \alpha (A_I) where
allow-all-id \equiv allow-pfun (id \circ Some)

**definition**
allow-all :: (\alpha \Rightarrow \text{unit}) (A_U) where
allow-all p = [allow ()]

**definition**
deny-all :: (\alpha \Rightarrow \text{unit}) (D_U) where
deny-all p = [deny ()]

... and resulting properties:

**lemma** A_I \bigoplus empty = A_I
(proof)

**lemma** A_f f \bigoplus empty = A_f f
(proof)

**lemma** allow-pfun empty = empty
(proof)

**lemma** allow-left-cancel : dom pf = UNIV \implies (allow-pfun pf) \bigoplus x = (allow-pfun pf)
(proof)

**lemma** deny-left-cancel : dom pf = UNIV \implies (deny-pfun pf) \bigoplus 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 \varnothing = \{ a. \varnothing a \neq \bot \}

whereas range is just an abbreviation for image:

 abbreviation range :: "(\alpha => \beta) => '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 dom (λx. [?f x]) = UNIV
- Map.dom_def dom ?m = {a. ?m a ≠ ⊥}
- Map.dom_empty dom ∅ = {} 
- Map.dom_eq_empty_conv (dom ?f = {}) = (?f = ∅)
- Map.dom_eq_singleton_conv (dom ?f = {?x}) = (∃ v. ?f = [?x ↦ v])
- Map.dom_fun_upd dom (?f(?x := ?y)) = (if ?y = ⊥ then dom ?f - {?x} else insert ?x (dom ?f))
- Map.dom_if dom (λx. if ?P x then ?f x else ?g x) = dom ?f ∩ {?x. ¬ ?P x} ∪ dom ??g ∩ {?x. ?P x}
- Map.dom_map_add dom (?n ⊕ ?m) = dom ?n ∪ dom ?m

However, some properties are specific to policy concepts:

**lemma** sub-ran : ran p ⊆ Allow ∪ Deny

**lemma** dom-allow-pfun [simp]: dom(allow-pfun f) = dom f

**lemma** dom-allow-all: dom(Af f) = UNIV

**lemma** dom-deny-pfun [simp]: dom(deny-pfun f) = dom f

**lemma** dom-deny-all: dom(Df f) = UNIV

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

16
lemma ran-allow-all: ran(\(A_f \ id\)) = Allow
⟨proof⟩

lemma ran-deny-pfun[simp]: ran(\(\text{deny-pfun } f\)) = \(\text{deny } \big(F (\text{ran } f)\big)\)
⟨proof⟩

lemma ran-deny-all: ran(\(D_f \ id\)) = Deny
⟨proof⟩

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

• Map.dom_map_add dom \( (?n \uplus ?m) = \text{dom } ?n \cup \text{dom } ?m \)
• Map.inj_on_map_add_dom inj-on \( (?m' \uplus ?m) (\text{dom } ?m) = \text{inj-on } ?m' (\text{dom } ?m) \)
• Map.map_add_comm dom ?m.0 \cap \text{dom } ?m.2 = \{\} \implies ?m.2 \uplus ?m.0 = ?m.0 \uplus ?m.2
• Map.map_add_dom_app_simps(1) ?m \in \text{dom } ?l.2 \implies (?l.2 \uplus ?l.1) ?m = ?l.2 \uplus ?m
• Map.map_add_dom_app_simps(2) ?m \notin \text{dom } ?l.1 \implies (?l.2 \uplus ?l.1) ?m = ?l.2 \uplus ?m
• Map.map_add_dom_app_simps(3) ?m \notin \text{dom } ?l.2 \implies (?l.2 \uplus ?l.1) ?m = ?l.1 \uplus ?m
• Map.map_add_upd_left ?m \notin \text{dom } ?e.2 \implies ?e.2 \uplus ?e.1(\?m \mapsto \?u.1) = (?e.2 \uplus ?e.1)(\?m \mapsto \?u.1)

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

definition dom-restrict :: \(\alpha \rightarrow \beta \rightarrow \gamma \rightarrow \gamma (\text{infixr } < 55)\)
where \( S \triangleleft p \equiv (\lambda x. \text{if } x \in S \text{ then } p x \text{ else } \bot) \)

lemma dom-dom-restrict[simp] : dom(S \triangleleft p) = S \cap \text{dom } p
⟨proof⟩

lemma dom-restrict-idem[simp] : (dom p) \triangleleft p = p
⟨proof⟩

lemma dom-restrict-inter[simp] : T \triangleleft S \triangleleft p = T \cap S \triangleleft p
⟨proof⟩

definition ran-restrict :: \(\alpha \rightarrow \beta \rightarrow \gamma \text{ decision set } \rightarrow \alpha \rightarrow \beta \text{ (infixr } > 55)\)

17
where \( p \triangleright S \equiv (\lambda x. \text{if } p \ x \in (\text{Some}'S) \text{ then } p \ x \text{ else } \bot) \)

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

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

**lemma** \( \text{ran-restrict} = \text{ran-restrict2} \)

⟨proof⟩

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

⟨proof⟩

**lemma** \( \text{ran-restrict-idem[simp]} : p \triangleright (\text{ran } p) = p \)

⟨proof⟩

**lemma** \( \text{ran-restrict-inter[simp]} : (p \triangleright S) \triangleright T = p \triangleright T \cap S \)

⟨proof⟩

**lemma** \( \text{ran-gen-A[simp]} : (\forall Ax. [P \ x]) \triangleright \text{Allow} = (\forall Ax. [P \ x]) \)

⟨proof⟩

**lemma** \( \text{ran-gen-D[simp]} : (\forall Dx. [P \ x]) \triangleright \text{Deny} = (\forall Dx. [P \ x]) \)

⟨proof⟩

**lemmas** \( \text{ElementaryPoliciesDefs} = \text{deny-pfun-def} \text{ allow-pfun-def} \text{ allow-all-fun-def} \text{ deny-all-fun-def} \text{ allow-all-id-def} \text{ deny-all-id-def} \text{ allow-all-def} \text{ deny-all-def} \text{ dom-restrict-def} \text{ ran-restrict-def} \)

end

**2.3 Sequential Composition**

**theory**

\( \text{SeqComposition} \)

**imports**

\( \text{ElementaryPolicies} \)

**begin**

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

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

fun flat-orA :: ('a decision) decision \Rightarrow ('a decision)
where flat-orA(allow(allow y)) = allow y
| flat-orA(allow(deny y)) = allow y
| flat-orA(deny(allow y)) = allow y
| flat-orA(deny(deny y)) = deny y

lemma flat-orA-deny[dest]: flat-orA x = deny y \Rightarrow x = deny(deny y)
⟨proof⟩

lemma flat-orA-allow[dest]: flat-orA x = allow y \Rightarrow x = allow(allow y)
\lor x = allow(deny y)
\lor x = deny(allow y)
⟨proof⟩

fun flat-orD :: ('a decision) decision \Rightarrow ('a decision)
where flat-orD(allow(allow y)) = allow y
| flat-orD(allow(deny y)) = deny y
| flat-orD(deny(allow y)) = deny y
| flat-orD(deny(deny y)) = deny y

lemma flat-orD-allow[dest]: flat-orD x = allow y \Rightarrow x = allow(allow y)
⟨proof⟩

lemma flat-orD-deny[dest]: flat-orD x = deny y \Rightarrow x = deny(deny y)
\lor x = allow(deny y)
\lor x = deny(allow y)
⟨proof⟩

fun flat-1 :: ('a decision) decision \Rightarrow ('a decision)
where flat-1(allow(allow y)) = allow y
| flat-1(allow(deny y)) = allow y
| flat-1(deny(allow y)) = deny y
| flat-1(deny(deny y)) = deny y

lemma flat-1-allow[dest]: flat-1 x = allow y \Rightarrow x = allow(allow y) \lor x = allow(deny y)
⟨proof⟩

lemma flat-1-deny[dest]: flat-1 x = deny y \Rightarrow x = deny(deny y) \lor x = deny(allow y)
fun flat-2 :: ('α decision) decision ⇒ ('α decision)
where flat-2(allow(allow y)) = allow y
    | flat-2(allow(deny y)) = deny y
    | flat-2(deny(allow y)) = allow y
    | flat-2(deny(deny y)) = deny y

lemma flat-2-allow[dest]: flat-2 x = allow y ⇒ x = allow(allow y) ∨ x = deny(allow y)
⟨proof⟩

lemma flat-2-deny[dest]: flat-2 x = deny y ⇒ x = deny(deny y) ∨ x = allow(deny y)
⟨proof⟩

2.3.2 Policy Composition

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

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

lemma lift-mt [simp]: lift ∅ = ∅
⟨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:

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

notation (xsymbols)
comp-orA (infixl o∨A 55)

lemma comp-orA-mt[simp]:p o∨A ∅ = ∅
⟨proof⟩
lemma mt-comp-orA[simp]: ∅ ◦_{\forall A} p = ∅ 
⟨proof⟩

definition
comp-orD :: \[ \beta \mapsto \gamma, \alpha \mapsto \beta \] \Rightarrow \alpha \mapsto \gamma \quad \text{(infixl} \ o'\text{-orD 55)} \quad \text{where}
\quad \text{where}
\quad p2 \ o\text{-orD} p1 \equiv (\text{map-option flat-orD}) \ o (\text{lift} \ p2 \ \circ_m \ p1)

notation \( \text{(xsymbols)} \)
\quad \text{comp-orD} \quad \text{(infixl} \ o_rD 55)

lemma comp-orD-mt[simp]: p \ o\text{-or} \ ∅ = ∅ 
⟨proof⟩

lemma mt-comp-orD[simp]: ∅ \ o\text{-orD} p = ∅ 
⟨proof⟩

definition
\quad \text{comp-1} :: \[ \beta \mapsto \gamma, \alpha \mapsto \beta \] \Rightarrow \alpha \mapsto \gamma \quad \text{(infixl} \ o'\text{-1 55)} \quad \text{where}
\quad \text{where}
\quad p2 \ o\text{-1} p1 \equiv (\text{map-option flat-1}) \ o (\text{lift} \ p2 \ \circ_m \ p1)

notation \( \text{(xsymbols)} \)
\quad \text{comp-1} \quad \text{(infixl} \ o_1 55)

lemma comp-1-mt[simp]: p \ o_1 \ ∅ = ∅ 
⟨proof⟩

lemma mt-comp-1[simp]: ∅ \ o_1 p = ∅ 
⟨proof⟩

definition
\quad \text{comp-2} :: \[ \beta \mapsto \gamma, \alpha \mapsto \beta \] \Rightarrow \alpha \mapsto \gamma \quad \text{(infixl} \ o'\text{-2 55)} \quad \text{where}
\quad \text{where}
\quad p2 \ o\text{-2} p1 \equiv (\text{map-option flat-2}) \ o (\text{lift} \ p2 \ \circ_m \ p1)

notation \( \text{(xsymbols)} \)
\quad \text{comp-2} \quad \text{(infixl} \ o_2 55)

lemma comp-2-mt[simp]: p \ o_2 \ ∅ = ∅ 
⟨proof⟩

lemma mt-comp-2[simp]: ∅ \ o_2 p = ∅ 
⟨proof⟩

end
2.4 Parallel Composition

theory
  ParallelComposition
imports
  ElementaryPolicies
begin

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

The new input pair will often contain repetitions, which can be reduced using the domain-restriction and domain-reduction operators. Using additional range-modifying operators such as $\nabla$, 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 :: \(\alpha \to' \beta, \gamma \to' \delta \) \(\Rightarrow \left(\alpha \times' \gamma \mapsto \beta \times' \delta\right)\) (infixr $\otimes_A$ 55)
where
  \(p1 \otimes_A p2 = (\lambda (x,y). (\text{case } p1 x \ of \allow  d1 \Rightarrow (\text{case } p2 y \ of \allow  d2 \Rightarrow \allow (d1,d2)) \mid \allow  \Rightarrow \bot))\)

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:\(p2 \otimes_A p1 = ((\lambda (x,y). (y,x)) o-f (p1 \otimes_A p2))) o (\lambda (a,b).(b,a))\)
\textbf{definition} prod-orD ::\[\alpha \mapsto \beta, \gamma \mapsto \delta\] \Rightarrow (\alpha \times \gamma \mapsto \beta \times \delta) \ (\text{infixr } \otimes_{\vee D} 55)
\textbf{where} p1 \otimes_{\vee D} p2 = 
(\lambda (x,y). \ (\text{case } p1 \ x \ of 
\begin{array}{ll}
[\text{allow } d1] \Rightarrow \ (\text{case } p2 \ y \ of 
\begin{array}{ll}
[\text{allow } d2] \Rightarrow [\text{allow}(d1, d2)] \\
[\text{deny } d2] \Rightarrow [\text{deny}(d1, d2)] \\
\bot \Rightarrow \bot
\end{array} \\
\end{array}) \\
| [\text{deny } d1] \Rightarrow \ (\text{case } p2 \ y \ of 
\begin{array}{ll}
[\text{allow } d2] \Rightarrow [\text{deny}(d1, d2)] \\
[\text{deny } d2] \Rightarrow [\text{deny}(d1, d2)] \\
\bot \Rightarrow \bot
\end{array}) \\
| \bot \Rightarrow \bot
\end{array})
\right)

\textbf{lemma} prod-orD-mt[simp]: p \otimes_{\vee D} \emptyset = \emptyset
\textbf{proof}

\textbf{lemma} mt-prod-orD[simp]: \emptyset \otimes_{\vee D} p = \emptyset
\textbf{proof}

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

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

\textbf{definition} prod-1 ::\[\alpha \mapsto \beta, \gamma \mapsto \delta\] \Rightarrow (\alpha \times \gamma \mapsto \beta \times \delta) \ (\text{infixr } \otimes_{1} 55)
\textbf{where} p1 \otimes_{1} p2 \equiv 
(\lambda (x,y). \ (\text{case } p1 \ x \ of 
\begin{array}{ll}
[\text{allow } d1] \Rightarrow \ (\text{case } p2 \ y \ of 
\begin{array}{ll}
[\text{allow } d2] \Rightarrow [\text{allow}(d1, d2)] \\
[\text{deny } d2] \Rightarrow [\text{deny}(d1, d2)] \\
\bot \Rightarrow \bot
\end{array} \\
\end{array}) \\
| [\text{deny } d1] \Rightarrow \ (\text{case } p2 \ y \ of 
\begin{array}{ll}
[\text{allow } d2] \Rightarrow [\text{deny}(d1, d2)] \\
[\text{deny } d2] \Rightarrow [\text{deny}(d1, d2)] \\
\bot \Rightarrow \bot
\end{array}) \\
| \bot \Rightarrow \bot
\end{array})
\right)

\textbf{lemma} prod-1-mt[simp]: p \otimes_{1} \emptyset = \emptyset
\textbf{proof}

\textbf{lemma} mt-prod-1[simp]: \emptyset \otimes_{1} p = \emptyset
\textbf{proof}
definition prod-2 :: \[\'\alpha \rightarrow \beta, \ '\gamma \mapsto \delta \] \Rightarrow \ (\'\alpha \times \ '\gamma \rightarrow \ '\beta \times \delta) \ (\text{infixr } \bigotimes \ 55) \\
where \ p1 \bigotimes \bigotimes 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{deny} \ (d1,d2)) \ |
\downarrow \Rightarrow \downarrow)) \]

lemma prod-2-mt[simp]: p \bigotimes \bigotimes \emptyset = \emptyset \\
⟨ \text{proof} \rangle 

lemma mt-prod-2[simp]: \emptyset \bigotimes \bigotimes p = \emptyset \\
⟨ \text{proof} \rangle 

definition prod-1-id :: \[\'\alpha \rightarrow \beta, \ '\alpha \mapsto \ '\gamma \] \Rightarrow \ (\'\alpha \rightarrow \ '\gamma) \ (\text{infixr } \bigotimes \bigotimes 1l \ 55) \\
where \ p \bigotimes \bigotimes 1l q = (p \bigotimes \bigotimes 1 q) \ o \ (\lambda x. \ (x,x)) 

lemma prod-1-id-mt[simp]: p \bigotimes \bigotimes 1l \emptyset = \emptyset \\
⟨ \text{proof} \rangle 

lemma mt-prod-1-id[simp]: \emptyset \bigotimes \bigotimes 1l p = \emptyset \\
⟨ \text{proof} \rangle 

definition prod-2-id :: \[\'\alpha \rightarrow \beta, \ '\alpha \mapsto \ '\gamma \] \Rightarrow \ (\'\alpha \rightarrow \ '\gamma) \ (\text{infixr } \bigotimes \bigotimes 2l \ 55) \\
where \ p \bigotimes \bigotimes 2l q = (p \bigotimes \bigotimes 2 q) \ o \ (\lambda x. \ (x,x)) 

lemma prod-2-id-mt[simp]: p \bigotimes \bigotimes 2l \emptyset = \emptyset \\
⟨ \text{proof} \rangle 

lemma mt-prod-2-id[simp]: \emptyset \bigotimes \bigotimes 2l p = \emptyset \\
⟨ \text{proof} \rangle 

2.4.2 Combinators for Transition Policies

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

definition parallel-map :: (\'\alpha \rightarrow \ '\beta) \Rightarrow \ (\'\delta \rightarrow \ '\gamma) \Rightarrow
\[ (\alpha \times \delta \rightarrow \beta \times \gamma) \text{ (infixr } \times_{M} 60) \]

where \( p1 \times_{M} p2 = (\lambda (x,y). \text{ case } p1 \text{ of } [d1] \Rightarrow (\text{case } p2 \text{ y of } [d2] \Rightarrow ([d1,d2]) \mid \bot \Rightarrow \bot)) \)

**definition** parallel-st :: \((i \times \sigma \rightarrow \sigma) \Rightarrow (i \times \sigma' \rightarrow \sigma') \Rightarrow (i \times \sigma \times \sigma' \rightarrow \sigma \times \sigma') \text{ (infixr } \times_{S} 60)\]

where \( p1 \times_{S} p2 = (p1 \times_{M} p2) \circ (\lambda (a,b,c). ((a,b),a,c)) \)

### 2.4.3 Range Splitting

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

**definition** comp-ran-split :: \([(\alpha \rightarrow \gamma) \times (\alpha \rightarrow \gamma), \delta \rightarrow \beta] \Rightarrow ([\delta \rightarrow \beta] \times [\beta \times \gamma]) \text{ (infixr } \times_{\nabla} 100)\]

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

An alternative characterisation of the operator is as follows:

**lemma** comp-ran-split-charn:

\( (f, g) \times_{\nabla} p = (\lambda \text{ x. case } p \text{ of } [allow y] \Rightarrow (\text{case } ((\text{fst } p) \text{ (snd x)}) \text{ of } \bot \Rightarrow \bot \mid [z] \Rightarrow \text{[allow (y,z)]}) \mid [deny y] \Rightarrow (\text{case } ((\text{snd } p) \text{ (snd x)}) \text{ of } \bot \Rightarrow \bot \mid [z] \Rightarrow \text{[deny (y,z)]}) \mid \bot \Rightarrow \bot) \)

### 2.4.4 Distributivity of the parallel combinators

**lemma** distr-or1-a: \((F = F1 \oplus F2) \Rightarrow ((N \times_{1} F) \circ f) = ((N \times_{1} F1) \circ f) \oplus ((N \times_{1} F2) \circ f)) \)

\{proof\}

**lemma** distr-or1: \((F = F1 \oplus F2) \Rightarrow ((g \circ f ((N \times_{1} F) \circ f)) = ((g \circ f ((N \times_{1} F1) \circ f)) \oplus ((g \circ f ((N \times_{1} F2) \circ f))))) \)

\{proof\}

**lemma** distr-or2-a: \((F = F1 \oplus F2) \Rightarrow ((N \times_{2} F) \circ f) = ((N \times_{2} F1) \circ f) \oplus ((N \times_{2} F2) \circ f)) \)

\{proof\}
proof

lemma distr-or2: \((F = F1 \oplus F2) \implies ((\text{r o-f } ((N \otimes_2 F) \circ f)) =\)
\((\text{r o-f } ((N \otimes_2 F1) \circ f)) \oplus (\text{r o-f } ((N \otimes_2 F2) \circ f))))\)
proof

lemma distr-orA: \((F = F1 \oplus F2) \implies ((\text{g o-f } ((N \otimes_A V A F) \circ f)) =\)
\(((\text{g o-f } ((N \otimes_A F1) \circ f)) \oplus (\text{g o-f } ((N \otimes_A F2) \circ f))))\)
proof

lemma distr-orD: \((F = F1 \oplus F2) \implies ((\text{g o-f } ((N \otimes_D V D F) \circ f)) =\)
\(((\text{g o-f } ((N \otimes_D F1) \circ f)) \oplus (\text{g o-f } ((N \otimes_D F2) \circ f))))\)
proof

lemma coerc-assoc: \((\text{r o-f } P) \circ d = \text{r o-f } (P \circ d)\)
proof

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

Comparing Policies

Policy p is more defined than q:

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

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

lemma strictly-more-vs-more: strictly-more-defined p q ⇒ more-defined p q
⟨proof⟩

Policy p is more permissive than q:

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

lemma more-permissive-refl : p ⊑ A p
⟨proof⟩

⟨proof⟩

Policy p is more rejective than q:

definition more-rejective :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ bool (infixl ⊑ D)
where p ⊑ D q = (∀ x. (case q x of ⌊deny y⌋ ⇒ (∃ z. (p x = ⌊deny z⌋)) |
| ⌊allow y⌋ ⇒ True |
| ⊥ ⇒ True))

lemma more-rejective-trans : p ⊑ D p' ⇒ p' ⊑ D p'' ⇒ p ⊑ D p''
⟨proof⟩

lemma more-rejective-refl : p ⊑ D p
⟨proof⟩

lemma A f f ⊑ A p
⟨proof⟩

lemma A I ⊑ A p
⟨proof⟩
2.5.2 Combined Data-Policy Refinement

definition policy-refinement ::
  (′a ↦→ ′b) ⇒ (′a ⇒ ′a) ⇒ (′b ⇒ ′b) ⇒ (′a ⇒ ′b) ⇒ bool
  (- ⊑. - [50,50,50,50]50)
where
  p ⊑ₘ aₘ,ₘ bₘ q ≡
  (∀ a. case p a of
    ⊥ ⇒ True
  | [allow y] ⇒ (∀ a′∈{x. aₘ x=a}. bₘ a′ = [allow b']
    ∧ absₘ b' = y)
  | [deny y] ⇒ (∀ a′∈{x. aₘ x=a}. bₘ a′ = [deny b']
    ∧ absₘ b' = y)

theorem polref-refl: p ⊑ₘ id, id p
⟨proof⟩

theorem polref-trans:
  assumes A: p ⊑ₘ P" f, g p'
  and B: p' ⊑ₘ P" f', g' p"
  shows p ⊑ₘ P o f" o g" o f' o g' p"
⟨proof⟩

2.5.3 Equivalence of Policies

Equivalence over domain D

definition p-eq-dom :: (′a ↦→ ′b) ⇒ ′a set ⇒ (′a ↦→ ′b) ⇒ bool
  (- ≈. - [60,60,60]60)
where
  p ≈ₘ D q = (∀ x∈D. p x = q x)

p and q have no conflicts

definition no-conflicts :: (′a ↦→ ′b) ⇒ (′a ↦→ ′b) ⇒ bool
where
  no-conflicts p q = (dom p = dom q ∧ (∀ x∈(dom p).
    (case p x of
    | [allow y] ⇒ (∃ x. q x = [allow x])
    | [deny y] ⇒ (∃ x. q x = [deny x]))))

lemma policy-eq:
  assumes p-over-qA: p ⊑ₘ A q
  and q-over-pA: q ⊑ₘ A p
  and p-over-qD: q ⊑ₘ D p
  and q-over-pD: p ⊑ₘ D q
  and dom-eq: dom p = dom q
  shows no-conflicts p q
⟨proof⟩
Miscellaneous

lemma dom-inter: \([\text{dom } p \cap \text{dom } q = \{\}; p x = \lfloor y \rfloor] \implies q x = \bot\) 

⟨proof⟩

lemma dom-eq: \(\text{dom } p \cap \text{dom } q = \{\} \implies p \oplus_A q = p \oplus_D q\) 

⟨proof⟩

lemma dom-split-alt-def : \((f, g) \Delta p = (\text{dom}(p \triangleright \text{Allow}) \triangle (A_f f)) \bigoplus (\text{dom}(p \triangleright \text{Deny}) \triangle (D_f g))\) 

⟨proof⟩

end

2.6 Policy Transformations

theory Normalisation

imports SeqComposition ParallelComposition

begin

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

2.6.1 Elementary Operators

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

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

definition list2policy :: \('a \mapsto 'b\) list ⇒ ('a ⇒ 'b) where
list2policy \(l = \text{foldr } (\lambda x y. (x \oplus y)) l \emptyset\)

Determine the position of element of a list.

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

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

fun applied-rule where
applied-rule \(C a (x # xs) = (if a \in \text{dom } (C x) then (Some x)\)
\begin{itemize}

- \( \text{applied-rule } C \ a \ [] = \text{None} \)

The following is used if the list is constructed backwards.

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

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

**Function** rm-MT-rules where
\[
\text{rm-MT-rules } C \ (x \# xs) = \begin{cases} 
\text{if } \text{dom} \ (C \ x) = \{\} & \text{then } \text{rm-MT-rules } C \ xs \\
\text{else } x \# (\text{rm-MT-rules } C \ xs) 
\end{cases}
\]
\[
|\text{rm-MT-rules } C \ [] = []
\]

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

**Function** none-MT-rules where
\[
\text{none-MT-rules } C \ (x \# xs) = (\text{dom} \ (C \ x) \neq \{\} \ \&\ \ (\text{none-MT-rules } C \ xs))
\]
\[
|\text{none-MT-rules } C \ [] = \text{True}
\]

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

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

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

**Lemma** none-MT-rules-vs-notMT: \( \text{none-MT-rules } C \ p \implies p \neq [] \implies \text{not-MT } C \ p \)

**Lemma** rmnMT: \( \text{none-MT-rules } C \ (\text{rm-MT-rules } C \ p) \)

**Lemma** rmnMT2: \( \text{none-MT-rules } C \ p \implies (\text{rm-MT-rules } C \ p) = p \)

**Lemma** nMTcharn: \( \text{none-MT-rules } C \ p = (\forall \ r \in \text{set } p. \ \text{dom} \ (C \ r) \neq \{\}) \)

**Lemma** nMTeqSet: \( \text{set } p = \text{set } s \implies \text{none-MT-rules } C \ p = \text{none-MT-rules } C \ s \)

**Lemma** notMTnMT: \( [a \in \text{set } p; \ \text{none-MT-rules } C \ p] \implies \text{dom} \ (C \ a) \neq \{\} \)

\end{itemize}
\begin{proof}

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

\begin{proof}

\textbf{lemma nMTtail: none-MT-rules C p \implies none-MT-rules C \((\text{tl p})\)}

\begin{proof}

\textbf{lemma not-MTimpnotMT[simp]: not-MT C p \implies p \neq []}

\begin{proof}

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

\begin{proof}

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

\begin{proof}

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

\begin{proof}

Next, a few theorems about applied\_rule:

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

\begin{proof}

\textbf{lemma mreq-end: \{applied-rule-rev C x b = Some r; applied-rule-rev C x c = Some r\} \implies applied-rule-rev C x \((a\#b)\) = applied-rule-rev C x \((a\#c)\)}

\begin{proof}

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

\begin{proof}

\textbf{lemma mreq-endNone: \{applied-rule-rev C x b = None; applied-rule-rev C x c = None\} \implies applied-rule-rev C x \((a\#b)\) = applied-rule-rev C x \((a\#c)\)}

\begin{proof}

\textbf{lemma mreq-end2: applied-rule-rev C x b = applied-rule-rev C x c \implies applied-rule-rev C x \((a\#b)\) = applied-rule-rev C x \((a\#c)\)}

\begin{proof}

\textbf{lemma mreq-end3: applied-rule-rev C x p \neq None \implies}

\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\[ \text{applied-rule-rev } C \ x \ (b \# \ p) = \text{applied-rule-rev } C \ x \ (p) \]

(\text{proof})

**lemma** mrNoneMT: \([ r \in \text{set } p; \text{applied-rule-rev } C \ x \ p = \text{None}] \implies x \notin \text{dom } (C \ r)\]

(\text{proof})

### 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.

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

An instance, as usual there are four of them.

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

**lemma** list2listNMT: \( x \neq [\] \implies \text{map sem } x \neq [\]
(\text{proof})

**lemma** two-conc: \( (\text{prod-list } x \ (y \# y s) \ p \ r \ d) = ((r \ o-f ((p \ x \ y) \ o \ d)) \# (\text{prod-list } x \ y s \ p \ r \ d)) \)
(\text{proof})

The following two invariants establish if the law of distributivity holds for a combinator and if an operator is strict regarding undefinedness.
definition \( \text{is-distr} \) where
\[
\text{is-distr } p = (\lambda g f. (\forall N P1 P2. (g o-f ((p N (P1 \oplus P2)) o f)) =
(g o-f ((p N P1) o f)) \oplus (g o-f ((p N P2) o f))))
\]

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

lemma \( \text{is-distr-orD} \): \( \text{is-distr} \) (\( \text{op} \otimes \bigvee D \)) \( d \) \( r \)
\langle proof \rangle

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

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

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

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

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

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

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

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

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

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

Some domain reasoning

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

33
lemma \textit{domSubstDistr2}: $\text{dom } A = \text{UNIV} \implies \text{dom } ((\lambda (x, y). x) \circ f (A \otimes_2 B) \circ (\lambda x. (x, x))) = \text{dom } B$

\langle proof \rangle

lemma \textit{domSubstDistrA}: $\text{dom } A = \text{UNIV} \implies \text{dom } ((\lambda (x, y). x) \circ f (A \otimes \nu A B) \circ (\lambda x. (x, x))) = \text{dom } B$

\langle proof \rangle

lemma \textit{domSubstDistrD}: $\text{dom } A = \text{UNIV} \implies \text{dom } ((\lambda (x, y). x) \circ f (A \otimes \nu D B) \circ (\lambda x. (x, x))) = \text{dom } B$

\langle proof \rangle

\end

\section{2.7 Policy Transformation for Testing}

theory
\textit{NormalisationTestSpecification}

imports
\textit{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 \textit{disjDom} where
\textit{disjDom} (x\#xs) = \((\forall y \in \text{(set xs)}. \text{dom } x \cap \text{dom } y = \{\}) \land \text{disjDom } xs\)
\text{disjDom } [] = \text{True}

fun \textit{PUTList} :: \('a \mapsto \text{bool}' \Rightarrow 'a \mapsto (\text{PUT } x \mapsto \text{bool}) \text{ list } \Rightarrow \text{bool}"

where
\textit{PUTList } PUT x (p\#ps) = \((x \in \text{dom } p \implies \text{PUT } x = p ) \land \text{PUTList } PUT x \text{ ps})
\text{PUTList } PUT x [] = \text{True}

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

\implies (\text{PUTList } PUT x \text{ PL } \implies \text{PUT } x = P x))

\langle proof \rangle

34
lemma PUTList-None: \( x \notin \text{dom} \ (\text{list2policy} \ \text{list}) \implies \text{PUTList} \ \text{PUT} \ x \ \text{list} \)
\langle \text{proof} \rangle

lemma PUTList-DomMT:
\((\forall \ y\in\text{set list. dom} \ a \cap \text{dom} \ y = \{\}) \implies x \in (\text{dom} \ a) \implies x \notin \text{dom} \ (\text{list2policy} \ \text{list}) \)
\langle \text{proof} \rangle

lemma distrPUTL2:
\( x \in \text{dom} \ P \implies (\text{list2policy} \ \text{PL}) \ x = P \ x \implies \text{disjDom} \ \text{PL} \implies (\text{PUT} \ x = P \ x) \implies \text{PUTList} \ \text{PUT} \ x \ \text{PL} \)
\langle \text{proof} \rangle

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

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

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

definition gatherDomain where \( \text{gatherDomain} \ p = (\text{gatherDomain-aux} \ (\text{butlast} \ p)) \)

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

definition disjDomGD where \( \text{disjDomGD} \ p = \text{disjDom} \ (\text{butlast} \ p) \)

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

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

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

\langle \text{proof} \rangle 

end

2.8 Putting Everything Together: UPF

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

lemmas UPFDefs = UPFCoreDefs ParallelDefs ElementaryPoliciesDefs
end
3 Example

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

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

3.1 Secure Service Specification

t``
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.
The security context stores all the existing LRs.

The user context stores the roles the users are in.

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

#### Health Records

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

- **datatype** `data` = `dummyContent`
- **datatype** `status` = `Open` | `Closed`
- **type-synonym** `entry-id` = `int`
- **type-synonym** `entry` = `status` × `user` × `data`
- **type-synonym** `SCR` = `(entry-id <!--[v]--> entry)`
- **type-synonym** `DB` = `patient` <!--[v]--> `SCR`

#### The Web Service API

The operations provided by the service:

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

**fun** `is-createSCR` where

- `is-createSCR (createSCR u r p) = True`
- `is-createSCR x = False`

**fun** `is-appendEntry` where

- `is-appendEntry (appendEntry u r e ei) = True`
- `is-appendEntry x = False`

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

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

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

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

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

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

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

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

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

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

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

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

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

fun allContentStatic where
allContentStatic \( (x \# xs) = (contentStatic x \land allContentStatic xs) \)
\[ |allContentStatic \emptyset = True \]

### 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

\[
get-entry S p e-id = \begin{cases} \bot & \text{if } S p \bot \bot \\ \lfloor Scr \rfloor & \Rightarrow Scr e-id \end{cases}
\]

**definition** \( userHasAccess:: user \Rightarrow entry \Rightarrow bool \) where

\( userHasAccess u e = ( (fst e) = Open \lor (fst (snd e) = u)) \)

**definition** \( readEntrySE :: SEPolicy \) where

\[
readEntrySE x = \begin{cases} \bot & \Rightarrow \bot \\ e & \Rightarrow (if (userHasAccess u e) \\
& \text{then } allow () \\
& \text{else } deny () ) \end{cases}
\]

**definition** \( deleteEntrySE :: SEPolicy \) where

\[
deleteEntrySE x = \begin{cases} \bot & \Rightarrow \bot \\ e & \Rightarrow (if (userHasAccess u e) \\
& \text{then } allow () \\
& \text{else } deny () ) \end{cases}
\]

**definition** \( editEntrySE :: SEPolicy \) where

\[
editEntrySE x = \begin{cases} \bot & \Rightarrow \bot \\ e & \Rightarrow (if (userHasAccess u e) \\
& \text{then } allow () \\
& \text{else } deny () ) \end{cases}
\]
\[ x \Rightarrow \bot \]

definition \text{SEPolicy} :: \text{SEPolicy where} \\
\text{SEPolicy} = \text{editEntrySE} \oplus \text{deleteEntrySE} \oplus \text{readEntrySE} \oplus A_U

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

Legitimate Relationships

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

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

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

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

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

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

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

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

\textbf{fun} \text{createFunPolicy} :: \text{FunPolicy where} \\
\text{createFunPolicy} ((\text{createSCR } u \ p), (D, S)) = (\text{if } p \in \text{dom } D \\
\text{then } [\text{deny ()}] \\
\text{else } [\text{allow ()}])

| createFunPolicy x = \bot
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 e 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 × u ↦→ 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) \cdot r = \text{Clerical} \land \text{is-removeLR } f}

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

3.1.4 The State Transitions and Output Function

State Transition

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

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

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

\text{Output}

\text{type-synonym Output = unit}

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

\[
\text{fun OpFailOutput :: } \text{Operation} \rightarrow \text{Output where} \\
\text{OpFailOutput } x = \lfloor \emptyset \rfloor
\]

\text{3.1.5 Combine All Parts}

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

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

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

\text{definition ST-Allow :: } \text{Operation } \times \text{DB} \times \Sigma \times \nu \rightarrow \text{Output } \times \text{DB} \times \Sigma \times \nu \\
\text{where ST-Allow} = ((\text{OpSuccessOutput} \otimes_M (\text{OpSuccessDB} \otimes_S \text{OpSuccessSigma} \\
\otimes_S \text{OpSuccessUC})) \\
\circ (\lambda (a,b,c). ((a),(a,b,c))))

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

\text{definition SE-LR-RBAC-ST-Policy :: } \text{Operation } \times \text{DB} \times \Sigma \times \nu \mapsto \text{Output } \times \text{DB} \times \Sigma \times \nu
where \( SE-LR-RBAC-ST-Policy = ((\lambda (x,y).y) \ o_1 ((ST-Allow,ST-Deny) \otimes \triangledown SE-LR-RBAC-Policy) \ o (\lambda x.(x,x))) \)

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

end

3.2 Instantiating Our Secure Service Example

theory
  ServiceExample
imports
  Service
begin

In the following, we briefly present an instantiation 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 \( UC0 :: \upsilon \) where
\( UC0 = \emptyset(alice\rightarrow\text{Nurse})(bob\rightarrow\text{ClinicalPractitioner})(charlie\rightarrow\text{Clerical}) \)

definition entry1 :: entry where
entry1 = (Open, alice, dummyContent)

definition entry2 :: entry where
entry2 = (Closed, bob, dummyContent)

definition entry3 :: entry where
entry3 = (Closed, alice, dummyContent)

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

definition \( Spine0 :: DB \) where
\[
Spine0 = \emptyset(patient1\mapsto SCR1)(patient2\mapsto SCR2)
\]

definition \( LR1 :: LR \) where
\[
LR1 = (\emptyset(1\mapsto \{alice\}))
\]

definition \( \Sigma0 :: \Sigma \) where
\[
\Sigma0 = (\emptyset(patient1\mapsto LR1))
\]

3.2.2 The Initial System State

definition \( \sigma0 :: DB \times \Sigma \times \upsilon \) where
\[
\sigma0 = (Spine0, \Sigma0, UC0)
\]

3.2.3 Basic Properties

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

(proof)

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

(proof)

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

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

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

\( \text{SE-LR-RBAC-ST-Policy-def map-add-def id-def LRsimps prod-2-def} \)

\( \text{RBACPolicy-def} \)

\( \text{SE-LR-Policy-def SEPolicy-def RBAC-def deleteEntrySE-def editEntrySE-def} \)

\( \text{readEntrySE-def } \sigma0\text{-def } \Sigma0\text{-def UC0-def patient1-def patient2-def LR1-def} \)

\( \text{alice-def bob-def charlie-def get-entry-def SE-LR-RBAC-Policy-def Allow-def} \)

\( \text{Deny-def dom-restrict-def policy-range-comp-def prod-orA-def prod-orD-def} \)

\( \text{ST-Allow-def ST-Deny-def Spine0-def SCR1-def SCR2-def entry1-def} \)

\( \text{entry2-def} \)

\( \text{entry3-def FunPolicy-def SE-LR-FUN-Policy-def o-def image-def UPFDefs} \)
lemma SE-LR-RBAC-Policy \((\text{createSCR} \; \text{alice} \; \text{Clerical} \; \text{patient1}) , \sigma_0) = \text{Some} \; (\text{deny} () ) \langle \text{proof} \rangle

lemma \text{exBool}[\text{simp}]: \exists \; a::\text{bool}. \; a \langle \text{proof} \rangle

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

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

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

lemma \((\sigma_0 \mid = (os \leftarrow \text{mbind} \; [(\text{createSCR} \; \text{alice} \; \text{Clerical} \; \text{patient1})] \; (\text{PolMon});
\quad (\text{return} \; (os = [(\text{deny} \; (\text{Out}) )]))) \langle \text{proof} \rangle

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

lemma SE-LR-RBAC-Policy \((\text{appendEntry} \; \text{alice} \; \text{Clerical} \; \text{patient1} \; \text{ei} \; \text{d}) , \sigma_0) = \lfloor \text{deny} () \rfloor \langle \text{proof} \rangle

end
4 Conclusion and Related Work

4.1 Related Work

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

4.2 Conclusion Future Work

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

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

5.1 Basic Monad Theory for Sequential Computations

theory Monads
imports Main
begin

5.1.1 General Framework for Monad-based Sequence-Test

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

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

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

State Exception Monads

type-synonym ('o, 'σ) MONSE = 'σ → ('o × 'σ)

definition bind-SE :: ('o,'σ)MONSE ⇒ ('o ⇒ ('o', 'σ)MONSE) ⇒ ('o','σ)MONSE
where  bind-SE f g = (λσ. case f σ of None ⇒ None
                   | Some (out, σ') ⇒ g out σ')

notation bind-SE (bindSE)
syntax  (xsymbols)
-bind-SE :: [pttrn,(′o,′σ)MON\textsubscript{SE},(′o,′σ)MON\textsubscript{SE}] ⇒ (′o,′σ)MON\textsubscript{SE}

(translations)

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

**definition** unit-SE :: ′o ⇒ (′o,′σ)MON\textsubscript{SE} 
**where** unit-SE e = (λσ. Some(e,σ))

**notation** unit-SE = (unit\textsubscript{SE})

**definition** fail-SE :: (′o,′σ)MON\textsubscript{SE} 
**where** fail-SE = (λσ. None)

**notation** fail-SE = (fail\textsubscript{SE})

**definition** assert-SE :: (′σ ⇒ bool) ⇒ (bool,′σ)MON\textsubscript{SE} 
**where** assert-SE P = (λσ. if P σ then Some(True,σ) else None)

**notation** assert-SE = (assert\textsubscript{SE})

**definition** assume-SE :: (′σ ⇒ bool) ⇒ (unit,′σ)MON\textsubscript{SE} 
**where** assume-SE P = (λσ. if ∃σ . P σ then Some((),SOME σ . P σ) else None)

**notation** assume-SE = (assume\textsubscript{SE})

**definition** if-SE :: [′σ ⇒ bool, (′α,′σ)MON\textsubscript{SE}, (′α,′σ)MON\textsubscript{SE}] ⇒ (′α,′σ)MON\textsubscript{SE} 
**where** if-SE c E F = (λσ. if c σ then E σ else F σ)

**notation** if-SE = (if\textsubscript{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 \textit{op}_1, \textit{op}_2, \ldots , \textit{op}_n with the inputs \textit{ι}_1, \textit{ι}_2, \ldots , \textit{ι}_n (outputs are treated analogously). Then we can encode for this interface the general input - type:

\[ \text{datatype in} = \textit{op}_1 :: \textit{t}_1 \mid \ldots \mid \textit{t}_n \]

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

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

\[
\text{datatype in } op_1 :: \iota_1 | \ldots | \iota_n
\]

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 ...

\[
\text{fun } mbind :: 'i list \Rightarrow ('i \Rightarrow ('o,'\sigma) \text{ MON}\_\text{SE}) \Rightarrow ('o list,'\sigma) \text{ MON}\_\text{SE}
\]

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

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

\[
\text{lemma } mbind\text{-unit } [\text{simp}]: \quad mbind [] f = (\text{return } [])
\]

\[
\begin{proof}
\end{proof}
\]

\[
\text{lemma } mbind\text{-nofailure } [\text{simp}]: \quad mbind S f \sigma \neq \text{None}
\]

\[
\begin{proof}
\end{proof}
\]
The fail-strict version of \( \text{mbind}' \) looks as follows:

\[
\begin{align*}
\text{fun} \quad \text{mbind}' :: & \ \ell \text{ list } \Rightarrow \ (' \ell \Rightarrow ('o, \sigma) \text{ MON}_SE) \Rightarrow ('o \text{ list}, \sigma) \text{ MON}_SE \\
\text{where} \quad \text{mbind}' [] \ iostep \ \sigma &= \ \text{Some}([], \sigma) \\
\quad \text{mbind}' (a\#H) \ iostep \ \sigma = & \\
\quad (\text{case} \ iostep \ a \ \sigma \ \text{of} & \\
\quad \quad \text{None} & \Rightarrow \text{None} \\
\quad \quad \text{Some} (\text{ out }, \sigma') & \Rightarrow (\text{case} \ \text{mbind} \ H \ iostep \ \sigma' \ \text{of} \\
\quad \quad \quad \text{None} & \Rightarrow \text{None} \ (* \ \text{fail-strict} \*) \\
\quad \quad \quad \text{Some} (\text{ outs }, \sigma'') & \Rightarrow \text{Some} (\text{out}\#\text{ outs }, \sigma'')))
\end{align*}
\]

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

\[
\begin{align*}
\text{definition} \quad \text{try-SE} :: & \ ('o, \sigma) \text{ MON}_SE \Rightarrow ('o \ \text{option}, \sigma) \text{ MON}_SE \\
\text{where} \quad \text{try-SE} \ ioprogs = & (\lambda \sigma. \ \text{case} \ ioprogs \ \sigma \ \text{of} \\
\quad \quad \text{None} & \Rightarrow \text{Some}(\text{None}, \sigma) \\
\quad \quad \text{Some} (\text{ outs }, \sigma') & \Rightarrow \text{Some}(\text{Some outs}, \sigma'))
\end{align*}
\]

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

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

\( \langle \text{proof} \rangle \)

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

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

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

\[
\begin{align*}
\text{lemma} \quad \text{malt-SE-mt} [\text{simp}]: & \prod_{SE} [] = \text{fail}_{SE} \\
\langle \text{proof} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{lemma} \quad \text{malt-SE-cons} [\text{simp}]: & \prod_{SE} (a \# S) = (a \cap_{SE} (\prod_{SE} S))
\end{align*}
\]

\[56\]
State-Backtrack Monads

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

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

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

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

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

definition \text{unit-SB} :: \text{'}\sigma \Rightarrow (\text{'}\sigma \times \text{'}\sigma) \text{ set} \Rightarrow (\text{'}\sigma) \text{MON}_\text{SB} ((\text{returns} \cdot) \text{ 8})

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

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

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

translations

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

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

\(\langle \text{proof} \rangle\)

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

\(\langle \text{proof} \rangle\)

lemma \textbind-assoc-SB : \((y := (x := m; k); h) = (x := m; (y := k; h))\)

\(\langle \text{proof} \rangle\)

State Backtrack Exception Monad

The following combination of the previous two Monad-Constructions allows for the semantic foundation of a simple generic assertion language in the style of Schirmer’s Simpl-Language or Rustan Leino’s Boogie-PL language. The key is to use the exceptional element None for violations of the assert-statement.

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

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

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

**Syntax**

- **bind-SBE** :: [pttrn, ('o', '\sigma')\text{MON}_{SBE}, ('o', '\sigma')\text{MON}_{SBE}] \Rightarrow ('o', '\sigma')\text{MON}_{SBE} \\
\text{translating} \ x :\equiv f; g \Rightarrow \text{CONST } \text{bind-SBE } f (\% x . g)

**Definition**

- **unit-SBE** :: ('o' \Rightarrow ('o', '\sigma')\text{MON}_{SBE} \ ((\text{returning -} \ 8)

where

- **assert-SBE** e = (\lambda \sigma. \text{if } e \ \sigma \text{ then Some}((((), \sigma)))

\text{else None})

**Notation**

- **assert-SBE** (assert_{SBE})

**Definition**

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

\text{else Some } \{}\}

**Notation**

- **assume-SBE** (assume_{SBE})

**Definition**

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

**Notation**

- **havoc-SBE** (havoc_{SBE})

**Lemma**

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

\langle \text{proof} \rangle

**Lemma**

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

\langle \text{proof} \rangle

**Lemmas**

- aux = \text{trans}[\text{OF } \text{HOL.neq-commute, OF } \text{Option.not-None-eq}]

**Lemma**

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

\langle \text{proof} \rangle

### 5.1.2 Valid Test Sequences in the State Exception Monad

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

**Definition**

- **valid-SB** : ('o' \Rightarrow (bool,'o') \text{MON}_{SBE} \Rightarrow bool \text{ (infix } \models 15)

where

- (\sigma \models m) = (m \ \sigma \neq \text{None } \land \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:**
\[ \sigma \models (os \leftarrow (mbind \ i s \ ioprog); \ return (length \ i s = length \ os)) \]
\[ \langle \text{proof} \rangle \]

**Lemma valid-true [simp]:**
\[ \sigma \models (s \leftarrow \ return \ x; \ return \ (P \ s))) = P \ x \]
\[ \langle \text{proof} \rangle \]
Recall mbind unit for the base case.

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

**Lemma valid-failure ′:**
\[ A \ \sigma = \text{None} \implies (\sigma \models ((s \leftarrow \ A \ ; \ M \ s))) \]
\[ \langle \text{proof} \rangle \]

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

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

**Lemma valid-success ′:**
\[ A \ \sigma = \text{Some} (b, \sigma') \implies (\sigma \models (s \leftarrow \ A \ ; \ M \ s)) = (\sigma' \models (M \ b)) \]
\[ \langle \text{proof} \rangle \]

**Lemma valid-both:**
\[ (\sigma \models (s \leftarrow \ \text{mbind} \ (a \# S) \ \ioprog \ ; \ return \ (P \ s))) = (\text{case} \ \ioprog \ a \ \sigma \ \text{of} \]
\[ \quad \text{None} \Rightarrow (\sigma \models (\text{return} \ (P []))) \]
\[ \quad \text{Some} (b, \sigma') \Rightarrow (\sigma' \models (s \leftarrow \ \text{mbind} \ S \ \ioprog \ ; \ return \ (P \ (b \# s)))) \]
lemma valid-propagate-1 [simp]: \( (\sigma | (\text{return } P)) = (P) \)

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

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

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

lemma valid-propagate-3 [simp]: \( (\sigma_0 | (\lambda \sigma. \text{Some } (f \sigma, \sigma))) = (f \sigma_0) \)

lemma valid-propagate-3': \( \neg (\sigma_0 | (\lambda \sigma. \text{None})) \)

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

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

lemma assert-disch3 : \( \neg P \sigma \implies \neg (\sigma | (\text{assert}_SE P)) \)

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

lemma assume-D : \( (\sigma | (x \leftarrow \text{assume}_SE P ; M x)) \implies \exists \sigma. (P \sigma \land \sigma | (M \text{False})) \)

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

**Lemma if-SE-D1**: \( P \sigma \implies (\sigma \models if_{SE} P B_1 B_2) = (\sigma \models B_1) \)

*Proof*

**Lemma if-SE-D2**: \( \neg P \sigma \implies (\sigma \models if_{SE} P B_1 B_2) = (\sigma \models B_2) \)

*Proof*

**Lemma if-SE-split-asm**: \( (\sigma \models if_{SE} P B_1 B_2) = ((P \sigma \land (\sigma \models B_1)) \lor (\neg P \sigma \land (\sigma \models B_2))) \)

*Proof*

**Lemma if-SE-split**: \( (\sigma \models if_{SE} P B_1 B_2) = ((P \sigma \implies (\sigma \models B_1)) \land (\neg P \sigma \implies (\sigma \models B_2))) \)

*Proof*

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

*Proof*

### 5.1.3 Valid Test Sequences in the State Exception Backtrack Monad

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

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

**Where** \( \sigma \models_{SBE} m \equiv (m \sigma \neq \text{None}) \)

This notation considers all non-failures as valid.

**Lemma assume-assert**: \( (\sigma \models_{SBE} (\_ \equiv \text{assume}_{SBE} P ; \text{assert}_{SBE} Q)) = (P \sigma \implies Q \sigma) \)

*Proof*

**Lemma assert-intro**: \( Q \sigma \implies (\sigma \models_{SBE} (\text{assert}_{SBE} Q)) \)

*Proof*

end
Bibliography


