The Unified Policy Framework (UPF)

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

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

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

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

March 17, 2025

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

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

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

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

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

2 The Unified Policy Framework (UPF)

2.1 The Core of the Unified Policy Framework (UPF)

theory
  UPFCore
  imports
    Monads
begin

2.1.1 Foundation

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

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

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

type-synonym $('\alpha,'\beta)$ *policy* = $'\alpha \rightharpoonup '\beta$ *decision* (**infixr** ‹|−>› *0*)

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

translations (*type*) $'\alpha$ |−> $'\beta$ <= (*type*) $'\alpha \rightharpoonup '\beta$ *decision*
type-notation *policy* (**infixr** ‹↦› *0*)

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

notation *None* (‹⊥›)
notation *Some* (‹⌊-⌋› *80*)

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

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

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

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

2.1.2 Policy Constructors

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

Furthermore, we add notation adopted to our problem domain:

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

```
syntax
  -policylet1  :: ['a, 'a] => policylet                (‹- /↦₊/ -›)
  -policylet2  :: ['a, 'a] => policylet                (‹- /↦₋/ -›)
               :: policylet => policylets              (‹-›)
  -Maplets     :: [policylet, policylets] => policylets (‹-,/ -›)
  -Maplets     :: [policylet, policylets] => policylets (‹-,/ -›)
   -MapUpd     :: ['a |-> 'b, policylets] => 'a |-> 'b  (‹-/'(-')› [900,0]900)
  -emptypolicy :: 'a |-> 'b                              (‹∅›)

syntax-consts
  -policylet1 ⇌ allow and
  -policylet2 ⇌ deny and
  -Maplets -MapUpd ⇌ fun-upd and
  -emptypolicy ⇌ Map.empty
translations
  -MapUpd m (-Maplets xy ms)    ⇌ -MapUpd (-MapUpd m xy) ms
  -MapUpd m (-policylet1 x y)  ⇌ m(x := CONST Some (CONST allow y))
```

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

Here are some lemmas essentially showing syntactic equivalences:

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

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

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

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

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

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

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

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

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

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

2.1.3 Override Operators

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

syntax
-policyoverride :: $['a \mapsto 'b,\ 'a \mapsto 'b] \Rightarrow 'a \mapsto 'b$ (**infixl** ‹$\bigoplus$› *100*)
syntax-consts
-policyoverride $\rightleftharpoons$ *map-add*
translations
$p \bigoplus q \rightleftharpoons q\ {+}{+}\ p$

Some elementary facts inherited from Map are:

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

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

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

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

```
definition override-A :: ['α↦'β, 'α↦'β] ⇒ 'α↦'β (infixl ‹++'-A› 100)
where  m2 ++-A m1 =
         (λx. (case m1 x of
                ⌊allow a⌋ ⇒  ⌊allow a⌋
              | ⌊deny a⌋  ⇒ (case m2 x of ⌊allow b⌋ ⇒ ⌊allow b⌋
                                          | - ⇒ ⌊deny a⌋)
              | ⊥ ⇒ m2 x)
          )

syntax
  -policyoverride-A  :: ['a ↦ 'b, 'a ↦ 'b] ⇒ 'a ↦ 'b (infixl ‹⊕_A› 100)
syntax-consts
  -policyoverride-A ⇌ override-A
translations
  p ⊕_A q ⇌ p ++-A q

lemma override-A-empty[simp]: p ⊕_A ∅ = p
  by(simp add:override-A-def)

lemma empty-override-A[simp]: ∅ ⊕_A p = p
  apply (rule ext)
  apply (simp add:override-A-def)
  subgoal for x
    apply (case-tac p x)
     apply (simp-all)
    subgoal for a
      apply (case-tac a)
       apply (simp-all)
      done
    done
  done
```

```
lemma override-A-assoc: p1 ⨁_A (p2 ⨁_A p3) = (p1 ⨁_A p2) ⨁_A p3
  by (rule ext, simp add: override-A-def split: decision.splits  option.splits)

definition override-D :: ['α↦'β, 'α↦'β] ⇒ 'α↦'β (infixl ‹++'-D› 100)
where m1 ++-D m2 =
          (λx. case m2 x of
                ⌊deny a⌋ ⇒ ⌊deny a⌋
             | ⌊allow a⌋ ⇒ (case m1 x of ⌊deny b⌋ ⇒ ⌊deny b⌋
                                   | - ⇒ ⌊allow a⌋)
             | ⊥ ⇒ m1 x
           )

syntax
  -policyoverride-D  :: ['a ↦ 'b, 'a ↦ 'b] ⇒ 'a ↦ 'b (infixl ‹⨁_D› 100)
syntax-consts
  -policyoverride-D ⇌ override-D
translations
  p ⨁_D q ⇌ p ++-D q

lemma override-D-empty[simp]: p ⨁_D ∅ = p
  by(simp add:override-D-def)

lemma empty-override-D[simp]: ∅ ⨁_D p = p
  apply (rule ext)
  apply (simp add:override-D-def)
  subgoal for x
    apply (case-tac p x, simp-all)
    subgoal for a
      apply (case-tac a, simp-all)
      done
    done
  done

lemma override-D-assoc: p1 ⨁_D (p2 ⨁_D p3) = (p1 ⨁_D p2) ⨁_D p3
  apply (rule ext)
  apply (simp add: override-D-def split: decision.splits  option.splits)
done
```

2.1.4 Coercion Operators

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

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

```
definition policy-range-comp :: ['β⇒'γ, 'α↦'β] ⇒ 'α ↦'γ   (infixl ‹o'-f› 55)
where
  f o-f p = (λx. case p x of
                   ⌊allow y⌋ ⇒ ⌊allow (f y)⌋
                 | ⌊deny y⌋ ⇒ ⌊deny (f y)⌋
                 | ⊥ ⇒ ⊥)

syntax
  -policy-range-comp :: ['β⇒'γ, 'α↦'β] ⇒ 'α ↦'γ (infixl ‹o_f› 55)
syntax-consts
  -policy-range-comp ⇌ policy-range-comp
translations
  p o_f q ⇌ p o-f q

lemma policy-range-comp-strict : f o_f ∅ = ∅
  apply (rule ext)
  apply (simp add: policy-range-comp-def)
  done
```

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

```
definition range-split :: [('β⇒'γ)×('β⇒'γ),'α ↦ 'β] ⇒ 'α ↦ 'γ
                          (infixr ‹∇› 100)
where (P) ∇ p = (λx. case p x of
                          ⌊allow y⌋ ⇒ ⌊allow ((fst P) y)⌋
                        | ⌊deny y⌋  ⇒ ⌊deny ((snd P) y)⌋
                        | ⊥         ⇒ ⊥)

lemma range-split-strict[simp]: P ∇ ∅ = ∅
  apply (rule ext)
  apply (simp add: range-split-def)
  done


lemma range-split-charn:
  (f,g) ∇ p = (λx. case p x of
                          ⌊allow x⌋ ⇒ ⌊allow (f x)⌋
                        | ⌊deny x⌋  ⇒ ⌊deny (g x)⌋
                        | ⊥         ⇒ ⊥)
  apply (simp add: range-split-def)
  apply (rule ext)
  subgoal for x
    apply (case-tac p x)
     apply (simp-all)
    subgoal for $a$
      apply ($case\text{-}tac\ a$)
       apply ($simp\text{-}all$)
      done
    done
  done
```

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

```
lemma $range\text{-}split\text{-}vs\text{-}range\text{-}compose$: $(f,f)\ \nabla\ p = f\ o_f\ p$
  by($simp\ add$: $range\text{-}split\text{-}charn\ policy\text{-}range\text{-}comp\text{-}def$)

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

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

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

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

```
definition $dom\text{-}split2a$ :: $[('\alpha \rightharpoonup '\gamma) \times ('\alpha \rightharpoonup '\gamma),'\alpha \mapsto '\beta] \Rightarrow '\alpha \mapsto '\gamma$            (infixr ‹$\Delta a$› $100$)
where $P\ \Delta a\ p = (\lambda x.\ case\ p\ x\ of$
```

$\lfloor allow\ y\rfloor \Rightarrow \lfloor allow\ (the\ ((fst\ P)\ x))\rfloor$
$\mid \lfloor deny\ y\rfloor \Rightarrow \lfloor deny\ (the\ ((snd\ P)\ x))\rfloor$
$\mid \bot \Rightarrow \bot)$

definition *dom-split2* :: $[('\alpha \Rightarrow '\gamma) \times ('\alpha \Rightarrow '\gamma), '\alpha \mapsto '\beta] \Rightarrow '\alpha \mapsto '\gamma$ (**infixr** ‹Δ› *100*)
where $P\ \Delta\ p = (\lambda x.\ case\ p\ x\ of$
$\lfloor allow\ y\rfloor \Rightarrow \lfloor allow\ ((fst\ P)\ x)\rfloor$
$\mid \lfloor deny\ y\rfloor \Rightarrow \lfloor deny\ ((snd\ P)\ x)\rfloor$
$\mid \bot \Rightarrow \bot)$

definition *range-split2* :: $[('\alpha \Rightarrow '\gamma) \times ('\alpha \Rightarrow '\gamma), '\alpha \mapsto '\beta] \Rightarrow '\alpha \mapsto ('\beta \times '\gamma)$ (**infixr** ‹∇2› *100*)
where $P\ \nabla 2\ p = (\lambda x.\ case\ p\ x\ of$
$\lfloor allow\ y\rfloor \Rightarrow \lfloor allow\ (y,(fst\ P)\ x)\rfloor$
$\mid \lfloor deny\ y\rfloor \Rightarrow \lfloor deny\ (y,(snd\ P)\ x)\rfloor$
$\mid \bot \Rightarrow \bot)$

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

definition *policy2MON* :: $('\iota \times '\sigma \mapsto 'o \times '\sigma) \Rightarrow ('\iota \Rightarrow ('o\ decision, '\sigma)\ MON_{SE})$
where $policy2MON\ p = (\lambda\ \iota\ \sigma.\ case\ p\ (\iota,\sigma)\ of$
$\lfloor (allow\ (outs,\sigma'))\rfloor \Rightarrow \lfloor (allow\ outs,\ \sigma')\rfloor$
$\mid \lfloor (deny\ (outs,\sigma'))\rfloor \Rightarrow \lfloor (deny\ outs,\ \sigma')\rfloor$
$\mid \bot \Rightarrow \bot)$

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

2.2 Elementary Policies

theory
ElementaryPolicies
imports
UPFCore
begin

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

2.2.1 The Core Policy Combinators: Allow and Deny Everything

definition
  deny-pfun $:: ('\alpha \rightharpoonup '\beta) \Rightarrow ('\alpha \mapsto '\beta)$ (‹*AllD*›)
  where
 deny-pfun pf $\equiv$ (λ *x. case pf x of*
      $\lfloor y \rfloor \Rightarrow \lfloor deny\ (y) \rfloor$
     $|\bot \Rightarrow \bot$)

definition
  allow-pfun $:: ('\alpha \rightharpoonup '\beta) \Rightarrow ('\alpha \mapsto '\beta)$ (‹*AllA*›)
  where
 allow-pfun pf $\equiv$ (λ *x. case pf x of*
      $\lfloor y \rfloor \Rightarrow \lfloor allow\ (y) \rfloor$
     $|\bot \Rightarrow \bot$)

syntax
 -allow-pfun $:: ('\alpha \rightharpoonup '\beta) \Rightarrow ('\alpha \mapsto '\beta)$ (‹A_p›)
syntax-consts
 -allow-pfun $\rightleftharpoons$ *allow-pfun*
translations
 $A_p\ f \rightleftharpoons AllA\ f$

syntax
 -deny-pfun $:: ('\alpha \rightharpoonup '\beta) \Rightarrow ('\alpha \mapsto '\beta)$ (‹D_p›)
syntax-consts
 -deny-pfun $\rightleftharpoons$ *deny-pfun*
translations
 $D_p\ f \rightleftharpoons AllD\ f$

notation
  deny-pfun (**binder** ‹$\forall$ *D*› *10*) **and**
  allow-pfun (**binder** ‹$\forall$ *A*› *10*)

lemma *AllD-norm*[*simp*]: *deny-pfun* (*id o* ($\lambda x.\ \lfloor x \rfloor$)) = ($\forall\, Dx.\ \lfloor x \rfloor$)
 by(*simp add:id-def comp-def*)

lemma *AllD-norm2*[*simp*]: *deny-pfun* (*Some o id*) = ($\forall\, Dx.\ \lfloor x \rfloor$)
 by(*simp add:id-def comp-def*)

lemma *AllA-norm*[*simp*]: *allow-pfun* (*id o Some*) = ($\forall\, Ax.\ \lfloor x \rfloor$)
 by(*simp add:id-def comp-def*)

lemma *AllA-norm2*[*simp*]: *allow-pfun* (*Some o id*) = ($\forall\, Ax.\ \lfloor x \rfloor$)

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

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

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

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

2.2.2 Common Instances

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

definition *deny-all-fun* :: $('\alpha \Rightarrow '\beta) \Rightarrow ('\alpha \mapsto '\beta)$ $(\langle D_f\rangle)$
  where *deny-all-fun f* ≡ *deny-pfun* (*Some o f*)

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

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

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

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

... and resulting properties:

lemma $A_I \bigoplus Map.empty = A_I$
  by *simp*

lemma $A_f\ f \bigoplus Map.empty = A_f\ f$
  by *simp*

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

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

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

lemma *deny-left-cancel* :*dom pf = UNIV* $\Longrightarrow$ (*deny-pfun pf*) $\bigoplus$ *x* = (*deny-pfun pf*)
  apply (*rule ext*)+
  by (*auto simp*: *deny-pfun-def option.splits*)

2.2.3 Domain, Range, and Restrictions

Since policies are essentially maps, we inherit the basic definitions for domain and range on Maps:
`Map.dom_def` : $dom\ ?m = \{a.\ ?m\ a \neq \bot\}$
whereas range is just an abrreviation for image:

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

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

- `Map.domD` $?a \in dom\ ?m \Longrightarrow \exists\, b.\ ?m\ ?a = \lfloor b \rfloor$
- `Map.domI` $?m\ ?a = \lfloor ?b \rfloor \Longrightarrow ?a \in dom\ ?m$
- `Map.domIff` $(?a \in dom\ ?m) = (?m\ ?a \neq \bot)$
- `Map.dom_const` $dom\ (\lambda x.\ \lfloor ?f\ x \rfloor) = UNIV$
- `Map.dom_def` $dom\ ?m = \{a.\ ?m\ a \neq \bot\}$
- `Map.dom_empty` $dom\ (\lambda x.\ \bot) = \{\}$
- `Map.dom_eq_empty_conv` $(dom\ ?f = \{\}) = (?f = (\lambda x.\ \bot))$

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

However, some properties are specific to policy concepts:

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

lemma *dom-allow-pfun* [*simp*]:*dom*(*allow-pfun f*) = *dom f*
  apply (*auto simp*: *allow-pfun-def*)
  subgoal for *x y*
    apply (*case-tac f x*, *simp-all*)
    done
  done

lemma *dom-allow-all*: *dom*(A_f *f*) = *UNIV*
  by(*auto simp*: *allow-all-fun-def o-def*)

lemma *dom-deny-pfun* [*simp*]:*dom*(*deny-pfun f*) = *dom f*
  apply (*auto simp*: *deny-pfun-def*)[*1*]
  apply (*case-tac f x*)
  apply (*simp-all*)
  done

lemma *dom-deny-all*: *dom*(D_f *f*) = *UNIV*
  by(*auto simp*: *deny-all-fun-def o-def*)

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

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

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

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

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

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

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

$?m1.0 \bigoplus ?m2.0$

- Map.map_add_dom_app_simps(1) $?m \in dom\ ?l2.0 \Longrightarrow (?l2.0 \bigoplus ?l1.0)\ ?m = ?l2.0\ ?m$
- Map.map_add_dom_app_simps(2) $?m \notin dom\ ?l1.0 \Longrightarrow (?l2.0 \bigoplus ?l1.0)\ ?m = ?l2.0\ ?m$
- Map.map_add_dom_app_simps(3) $?m \notin dom\ ?l2.0 \Longrightarrow (?l2.0 \bigoplus ?l1.0)\ ?m = ?l1.0\ ?m$
- Map.map_add_upd_left $?m \notin dom\ ?e2.0 \Longrightarrow ?e2.0 \bigoplus ?e1.0(?m \mapsto ?u1.0) = (?e2.0 \bigoplus ?e1.0)(?m \mapsto ?u1.0)$

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

definition *dom-restrict* :: $['\alpha\ set, '\alpha \mapsto '\beta] \Rightarrow '\alpha \mapsto '\beta$ (**infixr** ‹◁› *55*)
where $S \lhd p \equiv (\lambda x.\ if\ x \in S\ then\ p\ x\ else\ \bot)$

lemma *dom-dom-restrict*[*simp*] : $dom(S \lhd p) = S \cap dom\ p$
  apply (*auto simp*: *dom-restrict-def*)
  subgoal for *x y*
    apply (*case-tac* $x \in S$)
     apply (*simp-all*)
    done
  subgoal for *x y*
    apply (*case-tac* $x \in S$)
     apply (*simp-all*)
    done
  done

lemma *dom-restrict-idem*[*simp*] : $(dom\ p) \lhd p = p$
  apply (*rule ext*)
  apply (*auto simp*: *dom-restrict-def*
      dest: *neq-commute*[*THEN iffD1*,*THEN not-None-eq* [*THEN iffD1*]])
  done

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

definition *ran-restrict* :: $['\alpha \mapsto '\beta, '\beta\ decision\ set] \Rightarrow '\alpha \mapsto '\beta$ (**infixr** ‹▷› *55*)
where $p \rhd S \equiv (\lambda x.\ if\ p\ x \in (Some`S)\ then\ p\ x\ else\ \bot)$

```
definition ran-restrict2 :: [$'\alpha \mapsto '\beta$,$'\beta$ decision set] ⇒ $'\alpha \mapsto '\beta$ (infixr ‹▷2› 55)
where      p ▷2 S ≡ (λx. if (the (p x)) ∈ (S) then p x else ⊥)

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


lemma ran-ran-restrict[simp] : ran(p ▷ S) = S ∩ ran p
  by(auto simp: ran-restrict-def image-def ran-def)

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

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

lemma ran-gen-A[simp] : (∀Ax. ⌊P x⌋) ▷ Allow = (∀Ax. ⌊P x⌋)
  apply (rule ext)
  apply (auto simp: Allow-def ran-restrict-def)
  done

lemma ran-gen-D[simp] : (∀Dx. ⌊P x⌋) ▷ Deny = (∀Dx. ⌊P x⌋)
  apply (rule ext)
  apply (auto simp: Deny-def ran-restrict-def)
  done

lemmas ElementaryPoliciesDefs = deny-pfun-def allow-pfun-def allow-all-fun-def
deny-all-fun-def
                              allow-all-id-def deny-all-id-def allow-all-def deny-all-def
                              dom-restrict-def ran-restrict-def
```

end

2.3 Sequential Composition

```
theory
  SeqComposition
  imports
    ElementaryPolicies
begin
```

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

2.3.1 Flattening

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

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

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

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

fun *flat-orD* :: $('\alpha$ *decision*) *decision* $\Rightarrow$ $('\alpha$ *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* $\Longrightarrow$ *x* = *allow*(*allow y*)
  apply (*case-tac x*)
   apply (*rename-tac* α)
   apply (*case-tac* α, *simp-all*)[*1*]
  apply (*rename-tac* α)
  apply (*case-tac* α, *simp-all*)[*1*]
  done

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

fun *flat-1* :: $('\alpha$ *decision*) *decision* $\Rightarrow$ $('\alpha$ *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* $\Longrightarrow$ *x* = *allow*(*allow y*) $\vee$ *x* = *allow*(*deny y*)
  apply (*case-tac x*)
   apply (*rename-tac* α)
   apply (*case-tac* α, *simp-all*)[*1*]
  apply (*rename-tac* α)
  apply (*case-tac* α, *simp-all*)[*1*]
  done

lemma *flat-1-deny*[*dest*]: *flat-1 x* = *deny y* $\Longrightarrow$ *x* = *deny*(*deny y*) $\vee$ *x* = *deny*(*allow y*)
  apply (*case-tac x*)
   apply (*rename-tac* α)

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

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

lemma *flat-2-allow*[*dest*]: *flat-2 x* = *allow y* ⟹ *x* = *allow*(*allow y*) ∨ *x* = *deny*(*allow y*)

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

lemma *flat-2-deny*[*dest*]: *flat-2 x* = *deny y* ⟹ *x* = *deny*(*deny y*) ∨ *x* = *allow*(*deny y*)

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

2.3.2 Policy Composition

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

fun *lift* :: ($'\alpha \mapsto {'\beta}$) ⇒ ($'\alpha$ *decision* $\mapsto{'\beta}$ *decision*)
where *lift f* (*deny s*) = (*case f s of*
        $\lfloor y\rfloor \Rightarrow \lfloor$*deny y*$\rfloor$
        $\mid \bot \Rightarrow \bot$)
    | *lift f* (*allow s*) = (*case f s of*
        $\lfloor y\rfloor \Rightarrow \lfloor$*allow y*$\rfloor$
        $\mid \bot \Rightarrow \bot$)

lemma *lift-mt* [*simp*]: *lift* ∅ = ∅

```
  apply (rule ext)
  subgoal for x
```

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

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

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

notation
comp-orA (**infixl** ‹$\circ_{\vee A}$› *55*)

lemma *comp-orA-mt*[*simp*]:$p \circ_{\vee A} \emptyset = \emptyset$
by(*simp add*: *comp-orA-def*)

lemma *mt-comp-orA*[*simp*]:$\emptyset \circ_{\vee A} p = \emptyset$
by(*simp add*: *comp-orA-def*)

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

notation
comp-orD (**infixl** ‹$\circ_o rD$› *55*)

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

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

definition
comp-1 :: $['\beta \mapsto '\gamma, '\alpha \mapsto '\beta] \Rightarrow '\alpha \mapsto '\gamma$ (**infixl** ‹*o'-1*› *55*) **where**
p2 o-1 p1 $\equiv$ (*map-option flat-1*) *o* (*lift p2* $\circ_m$ *p1*)

notation
comp-1 (**infixl** ‹$\circ_1$› *55*)

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

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

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

notation
  comp-2 (**infixl** ‹$\circ_2$› *55*)

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

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

end

2.4 Parallel Composition

theory
  ParallelComposition
  imports
    ElementaryPolicies
begin

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

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

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

2.4.1 Parallel Combinators: Foundations

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

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

```
    (λ(x,y). (case p1 x of
        ⌊allow d1⌋ ⇒(case p2 y of
                              ⌊allow d2⌋ ⇒ ⌊allow(d1,d2)⌋
                            | ⌊deny d2⌋  ⇒ ⌊allow(d1,d2)⌋
                            | ⊥ ⇒ ⊥)
      | ⌊deny d1⌋⇒(case p2 y of
                              ⌊allow d2⌋ ⇒ ⌊allow(d1,d2)⌋
                            | ⌊deny d2⌋  ⇒ ⌊deny (d1,d2)⌋
                            | ⊥ ⇒ ⊥)
      | ⊥ ⇒ ⊥))
```

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

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

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

definition *prod-orD* ::[′α ↦ ′β, ′γ ↦ ′δ] ⇒ (′α × ′γ ↦ ′β × ′δ) (**infixr** ‹$\bigotimes_{\vee D}$› *55*)
where *p1* $\bigotimes_{\vee D}$ *p2* =

```
    (λ(x,y). (case p1 x of
        ⌊allow d1⌋ ⇒(case p2 y of
                              ⌊allow d2⌋ ⇒ ⌊allow(d1,d2)⌋
                            | ⌊deny d2⌋  ⇒ ⌊deny(d1,d2)⌋
                            | ⊥ ⇒ ⊥)
      | ⌊deny d1⌋⇒(case p2 y of
                              ⌊allow d2⌋ ⇒ ⌊deny(d1,d2)⌋
                            | ⌊deny d2⌋  ⇒ ⌊deny (d1,d2)⌋
                            | ⊥ ⇒ ⊥)
      | ⊥ ⇒ ⊥))
```

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

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

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

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

definition *prod-1* :: $['\alpha \mapsto '\beta,\ '\gamma \mapsto '\delta] \Rightarrow ('\alpha \times '\gamma \mapsto '\beta \times '\delta)$ (**infixr** ‹$\bigotimes_1$› *55*)
  where $p1 \bigotimes_1 p2 \equiv$
```
      (λ(x,y). (case p1 x of
            ⌊allow d1⌋⇒(case p2 y of
                              ⌊allow d2⌋ ⇒ ⌊allow(d1,d2)⌋
                            | ⌊deny d2⌋  ⇒ ⌊allow(d1,d2)⌋
                            | ⊥ ⇒ ⊥)
          | ⌊deny d1⌋ ⇒(case p2 y of
                              ⌊allow d2⌋ ⇒ ⌊deny(d1,d2)⌋
                            | ⌊deny d2⌋  ⇒ ⌊deny(d1,d2)⌋
                            | ⊥ ⇒ ⊥)
          |⊥ ⇒ ⊥))
```

lemma *prod-1-mt*[*simp*]:$p \bigotimes_1 \emptyset = \emptyset$
  apply (*rule ext*)
  apply (*simp add*: *prod-1-def*)
  apply (*auto*)[*1*]
  apply (*simp split*: *option.splits decision.splits*)
  done

lemma *mt-prod-1*[*simp*]:$\emptyset \bigotimes_1 p = \emptyset$
  apply (*rule ext*)

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

definition prod-2 :: ['α↦'β, 'γ ↦'δ] ⇒ ('α×'γ ↦ 'β×'δ) (infixr ‹⨂₂› 55)
  where p1 ⨂₂ p2 ≡
     (λ(x,y). (case p1 x of
          ⌊allow d1⌋ ⇒(case p2 y of
                              ⌊allow d2⌋ ⇒ ⌊allow(d1,d2)⌋
                            | ⌊deny d2⌋  ⇒ ⌊deny (d1,d2)⌋
                            | ⊥ ⇒ ⊥)
        | ⌊deny d1⌋⇒(case p2 y of
                              ⌊allow d2⌋ ⇒ ⌊allow(d1,d2)⌋
                            | ⌊deny d2⌋ ⇒  ⌊deny (d1,d2)⌋
                            | ⊥ ⇒ ⊥)
        |⊥ ⇒⊥))

lemma prod-2-mt[simp]:p ⨂₂ ∅ = ∅
  apply (rule ext)
  apply (simp add: prod-2-def)
  apply (auto)[1]
  apply (simp split: option.splits decision.splits)
  done

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

definition prod-1-id ::['α↦'β, 'α↦'γ] ⇒ ('α ↦ 'β×'γ) (infixr ‹⨂₁ᵢ› 55)
  where p ⨂₁ᵢ q = (p ⨂₁ q) o (λx. (x,x))

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

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

definition prod-2-id ::['α↦'β, 'α↦'γ] ⇒ ('α ↦ 'β×'γ) (infixr ‹⨂₂ᵢ› 55)
  wherep ⨂₂ᵢ q = (p ⨂₂ q) o (λx. (x,x))
```

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

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

2.4.2 Combinators for Transition Policies

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

definition *parallel-map* :: $('\alpha \rightharpoonup '\beta) \Rightarrow ('\delta \rightharpoonup '\gamma) \Rightarrow$
$('\alpha \times '\delta \rightharpoonup '\beta \times '\gamma)$ (**infixr** ‹$\bigotimes_M$› *60*)
  where $p1 \bigotimes_M p2 = (\lambda\ (x,y).\ case\ p1\ x\ of\ \lfloor d1 \rfloor \Rightarrow$
$(case\ p2\ y\ of\ \lfloor d2 \rfloor \Rightarrow \lfloor (d1,d2) \rfloor$
$\mid \bot \Rightarrow \bot)$
$\mid \bot \Rightarrow \bot)$

definition *parallel-st* :: $('i \times '\sigma \rightharpoonup '\sigma) \Rightarrow ('i \times '\sigma' \rightharpoonup '\sigma') \Rightarrow$
$('i \times '\sigma \times '\sigma' \rightharpoonup '\sigma \times '\sigma')$ (**infixr** ‹$\bigotimes_S$› *60*)
where
  $p1 \bigotimes_S p2 = (p1 \bigotimes_M p2)\ o\ (\lambda\ (a,b,c).\ ((a,b),a,c))$

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* :: $[('\alpha \rightharpoonup '\gamma) \times ('\alpha \rightharpoonup '\gamma), 'd \mapsto '\beta] \Rightarrow ('d \times '\alpha) \mapsto ('\beta \times '\gamma)$
(**infixr** ‹$\bigotimes_\nabla$› *100*)
where $P \bigotimes_\nabla p \equiv \lambda x.\ case\ p\ (fst\ x)\ of$
$\lfloor allow\ y \rfloor \Rightarrow (case\ ((fst\ P)\ (snd\ x))\ of\ \bot \Rightarrow \bot \mid \lfloor z \rfloor \Rightarrow \lfloor allow$
$(y,z) \rfloor)$
$\mid \lfloor deny\ y \rfloor \Rightarrow\ (case\ ((snd\ P)\ (snd\ x))\ of\ \bot \Rightarrow \bot \mid \lfloor z \rfloor \Rightarrow \lfloor deny$
$(y,z) \rfloor)$
$\mid \bot \Rightarrow \bot$

An alternative characterisation of the operator is as follows:

lemma *comp-ran-split-charn*:
  $(f,\ g)\ \bigotimes_\nabla\ p = ($

```
$(((p \ \triangleright \mathit{Allow}) \bigotimes_{\vee A} (A_p\ f)) \bigoplus$
 $((p \ \triangleright \mathit{Deny}) \bigotimes_{\vee A} (D_p\ g))))$
  apply (rule ext)
  apply (simp add: comp-ran-split-def map-add-def o-def ran-restrict-def image-def
      Allow-def Deny-def dom-restrict-def prod-orA-def
      allow-pfun-def deny-pfun-def
      split:option.splits decision.splits)
  apply (auto)
  done
```

2.4.4 Distributivity of the parallel combinators

```
lemma distr-or1-a: $(F = F1 \bigoplus F2) \Longrightarrow (((N \bigotimes_1 F)\ o\ f) =$
            $(((N \bigotimes_1 F1)\ o\ f) \bigoplus ((N \bigotimes_1 F2)\ o\ f)))$
  apply (rule ext)
  apply (simp add:  prod-1-def map-add-def
      split: decision.splits option.splits)
  subgoal for x
    apply (case-tac f x)
    apply (simp-all add: prod-1-def map-add-def
        split: decision.splits option.splits)
    done
  done

lemma distr-or1: $(F = F1 \bigoplus F2) \Longrightarrow ((g\ o\text{-}f\ ((N \bigotimes_1 F)\ o\ f)) =$
            $((g\ o\text{-}f\ ((N \bigotimes_1 F1)\ o\ f)) \bigoplus (g\ o\text{-}f\ ((N \bigotimes_1 F2)\ o\ f))))$
  apply (rule ext)+
  apply (simp add: prod-1-def map-add-def policy-range-comp-def
      split: decision.splits option.splits)
  subgoal for x
    apply (case-tac f x)
    apply (simp-all add: prod-1-def map-add-def
        split: decision.splits option.splits)
    done
  done

lemma distr-or2-a: $(F = F1 \bigoplus F2) \Longrightarrow (((N \bigotimes_2 F)\ o\ f) =$
            $(((N \bigotimes_2 F1)\ o\ f) \bigoplus ((N \bigotimes_2 F2)\ o\ f)))$
  apply (rule ext)
  apply (simp add: prod-2-id-def prod-2-def map-add-def
      split: decision.splits option.splits)
  subgoal for x
    apply (case-tac f x)
    apply (simp-all add: prod-2-def map-add-def
      split: decision.splits option.splits)
    done
  done

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

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

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

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

  done

lemmas *ParallelDefs* = *prod-orA-def* *prod-orD-def* *prod-1-def* *prod-2-def* *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 $\quad$ *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 $\quad$ *more-defined* $p\ q = (dom\ q \subseteq dom\ p)$

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

lemma *strictly-more-vs-more*: *strictly-more-defined* $p\ q \Longrightarrow$ *more-defined* $p\ q$
  unfolding *more-defined-def strictly-more-defined-def*
  by *auto*

Policy p is more permissive than q:

definition *more-permissive* :: $('a \mapsto 'b) \Rightarrow ('a \mapsto 'b) \Rightarrow bool$ (**infixl** ‹$\sqsubseteq_A$› *60*)
where $p \sqsubseteq_A q = (\forall\ x.\ (case\ q\ x\ of\ \lfloor allow\ y \rfloor \Rightarrow (\exists\ z.\ (p\ x = \lfloor allow\ z \rfloor))$
$\qquad\qquad\qquad | \lfloor deny\ y \rfloor \Rightarrow True$
$\qquad\qquad\qquad | \bot \qquad \Rightarrow True))$

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

lemma more-permissive-trans : $p \sqsubseteq_A p' \Longrightarrow p' \sqsubseteq_A p'' \Longrightarrow p \sqsubseteq_A p''$
  unfolding more-permissive-def
  apply(auto split : option.split decision.split)
  subgoal for $x$ $y$
    apply(erule-tac $x = x$ and
        $P = \lambda x.$ case $p''$ $x$ of $\bot \Rightarrow$ True
                                   | $\lfloor allow\ y \rfloor \Rightarrow \exists z.\ p'\ x = \lfloor allow\ z \rfloor$
                                   | $\lfloor deny\ y \rfloor \Rightarrow$ True in allE)
    apply(simp, elim exE)
    by(erule-tac $x = x$ in allE, simp)
  done
```

Policy p is more rejective than q:

```
definition more-rejective :: ('a ↦ 'b) ⇒ ('a ↦ 'b) ⇒ bool (infixl ‹$\sqsubseteq_D$› 60)
  where  $p \sqsubseteq_D q = (\forall\ x.\ ($case q x of $\lfloor deny\ y \rfloor\ \Rightarrow (\exists\ z.\ (p\ x = \lfloor deny\ z \rfloor))$
                              | $\lfloor allow\ y \rfloor \Rightarrow$ True
                              | $\bot$          $\Rightarrow$ True))


lemma more-rejective-trans : $p \sqsubseteq_D p' \Longrightarrow p' \sqsubseteq_D p'' \Longrightarrow p \sqsubseteq_D p''$
  unfolding more-rejective-def
  apply(auto split : option.split decision.split)
  subgoal for $x$ $y$
    apply(erule-tac $x = x$ and
        $P = \lambda x.$ case $p''$ $x$ of $\bot \Rightarrow$ True
                                   | $\lfloor allow\ y \rfloor \Rightarrow$ True
                                   | $\lfloor deny\ y \rfloor \Rightarrow \exists z.\ p'\ x = \lfloor deny\ z \rfloor$ in allE)
    apply(simp, elim exE)
    by(erule-tac $x = x$ in allE, simp)
  done


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


lemma $A_f\ f \sqsubseteq_A p$
```

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

lemma $A_I \sqsubseteq_A p$
  unfolding more-permissive-def allow-all-fun-def allow-pfun-def allow-all-id-def
  by(auto split: option.split decision.split)
```

2.5.2 Combined Data-Policy Refinement

```
definition policy-refinement ::
  $('a \mapsto 'b) \Rightarrow ('a' \Rightarrow 'a) \Rightarrow ('b' \Rightarrow 'b) \Rightarrow ('a' \mapsto 'b') \Rightarrow bool$
  $(\langle\text{-} \sqsubseteq_{\text{-},\text{-}} \text{-}\rangle\ [50,50,50,50]50)$
  where    $p \sqsubseteq_{abs_a,abs_b} q \equiv$
          $(\forall\ a.\ case\ p\ a\ of$
                $\bot \Rightarrow True$
              $|\ \lfloor allow\ y \rfloor \Rightarrow (\forall\ a' \in \{x.\ abs_a\ x=a\}.$
                                $\exists\ b'.\ \ q\ a' = \lfloor allow\ b' \rfloor$
                                      $\wedge\ abs_b\ b' = y)$
              $|\ \lfloor deny\ y \rfloor \Rightarrow (\forall\ a' \in \{x.\ abs_a\ x=a\}.$
                                $\exists\ b'.\ \ q\ a' = \lfloor deny\ b' \rfloor$
                                      $\wedge\ abs_b\ b' = y))$

theorem polref-refl: $p \sqsubseteq_{id,id} p$
  unfolding policy-refinement-def
  by(auto split: option.split decision.split)

theorem polref-trans:
  assumes A: $p \sqsubseteq_{f,g} p'$
    and    B: $p' \sqsubseteq_{f',g'} p''$
  shows   $p \sqsubseteq_{f\ o\ f',g\ o\ g'} p''$
  apply(insert A B)
  unfolding policy-refinement-def
  apply(auto split: option.split decision.split simp: o-def)[1]
  subgoal for $a\ a'$
    apply(erule-tac x=f (f' a') in allE, simp)[1]
    apply(erule-tac x=f' a' in allE, auto)[1]
    apply(erule-tac x= (f' a') in allE, auto)[1]
    done
  subgoal for $a\ a'$
    apply(erule-tac x=f (f' a') in allE, simp)[1]
    apply(erule-tac x=f' a' in allE, auto)[1]
    apply(erule-tac x= (f' a') in allE, auto)[1]
    done
  done
```

2.5.3 Equivalence of Policies

Equivalence over domain D

definition *p-eq-dom* :: $('a \mapsto 'b) \Rightarrow 'a\ set \Rightarrow ('a \mapsto 'b) \Rightarrow bool$ (‹- ≈₋ -› [60,60,60]60)
  where $p \approx_D q = (\forall x{\in}D.\ p\ x = q\ x)$

p and q have no conflicts

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

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

Miscellaneous

lemma *dom-inter*: $[\![dom\ p \cap dom\ q = \{\};\ p\ x = \lfloor y\rfloor]\!] \Longrightarrow q\ x = \bot$
  by (*auto*)

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

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

end

2.6 Policy Transformations

theory
  Normalisation
  imports
    SeqComposition
    ParallelComposition
begin

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

2.6.1 Elementary Operators

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

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

definition $\mathit{list2policy}::('a \mapsto 'b)\ \mathit{list} \Rightarrow ('a \mapsto 'b)$ **where**
  $\mathit{list2policy}\ l = \mathit{foldr}\ (\lambda\ x\ y.\ (x \bigoplus\ y))\ l\ \emptyset$

Determine the position of element of a list.

fun $\mathit{position} :: '\alpha \Rightarrow '\alpha\ \mathit{list} \Rightarrow\ \mathit{nat}$ **where**
  $\mathit{position}\ a\ []\qquad = 0$
$|(\mathit{position}\ a\ (x\#xs)) = (\mathit{if}\ a = x\ \mathit{then}\ 1\ \mathit{else}\ (\mathit{Suc}\ (\mathit{position}\ a\ xs)))$

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

fun *applied-rule* **where**
  $\mathit{applied\text{-}rule}\ C\ a\ (x\#xs) = (\mathit{if}\ a \in \mathit{dom}\ (C\ x)\ \mathit{then}\ (\mathit{Some}\ x)$
$\qquad\qquad\qquad\qquad \mathit{else}\ (\mathit{applied\text{-}rule}\ C\ a\ xs))$
$|\mathit{applied\text{-}rule}\ C\ a\ []\quad = \mathit{None}$

The following is used if the list is constructed backwards.

definition *applied-rule-rev* **where**
  $\mathit{applied\text{-}rule\text{-}rev}\ C\ a\ x =\ \mathit{applied\text{-}rule}\ C\ a\ (\mathit{rev}\ x)$

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

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

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

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

fun *none-MT-rules* **where**
  none-MT-rules C $(x\#xs)$ = (*dom* $(C\ x) \neq \{\} \wedge$ (*none-MT-rules C xs*))
|*none-MT-rules C* [] = *True*

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

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

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

lemma *none-MT-rules-vs-notMT*: *none-MT-rules C p* $\Longrightarrow p \neq [] \Longrightarrow$ *not-MT C p*
  apply (*induct p*)
   apply (*simp-all*)
  done

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

lemma *rmnMT2*: *none-MT-rules C p* $\Longrightarrow$ (*rm-MT-rules C p*) = *p*
  apply (*induct p*)
   apply (*simp-all*)
  done

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

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

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

```
  done

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

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

lemma not-MTimpnotMT[simp]: not-MT C p ⟹ p ≠ []
  apply (auto)
  done

lemma SR3nMT: ¬ not-MT C  p ⟹ rm-MT-rules C p = []
  apply (induct p)
   apply (auto simp: if-splits)
  done

lemma NMPcharn: ⟦a ∈ set p; dom (C a) ≠ {}⟧ ⟹ not-MT C  p
  apply (induct p)
   apply (auto simp: if-splits)
  done

lemma NMPrm: not-MT C  p ⟹ not-MT C (rm-MT-rules C p)
  apply (induct p)
   apply (simp-all)
  done
```

Next, a few theorems about applied_rule:

```
lemma mrconc: applied-rule-rev C x p = Some a ⟹ applied-rule-rev C x (b#p) =
Some a
proof (induct p rule: rev-induct)
  case Nil show ?case using Nil
    by (simp add: applied-rule-rev-def)
next
  case (snoc xs x) show ?case using snoc
    apply (simp add: applied-rule-rev-def if-splits)
    by (metis option.inject)
qed

lemma mreq-end: ⟦applied-rule-rev C x b = Some r; applied-rule-rev C x c = Some r⟧
```

```
⟹
 applied-rule-rev C x (a#b) = applied-rule-rev C x (a#c)
  by (simp add: mrconc)

lemma mrconcNone: applied-rule-rev C x p = None ⟹
                            applied-rule-rev C x (b#p) = applied-rule-rev C x [b]
proof (induct p rule: rev-induct)
  case Nil show ?case
    by (simp add: applied-rule-rev-def)
next
  case (snoc ys y) show ?case using snoc
  proof (cases x ∈ dom (C ys))
    case True show ?thesis using True snoc
      by (auto simp: applied-rule-rev-def)
  next
    case False show ?thesis using False snoc
      by (auto simp: applied-rule-rev-def)
  qed
qed

lemma mreq-endNone: ⟦applied-rule-rev C x b = None; applied-rule-rev C x c = None⟧
⟹
     applied-rule-rev C x (a#b) = applied-rule-rev C x (a#c)
  by (metis mrconcNone)

lemma mreq-end2: applied-rule-rev C x b = applied-rule-rev C x c ⟹
     applied-rule-rev C x (a#b) = applied-rule-rev C x (a#c)
  apply (case-tac applied-rule-rev C x b = None)
   apply (auto intro: mreq-end mreq-endNone)
  done

lemma mreq-end3: applied-rule-rev C x p ≠ None ⟹
                  applied-rule-rev C x (b # p) = applied-rule-rev C x (p)
  by (auto simp: mrconc)

lemma mrNoneMT: ⟦r ∈ set p; applied-rule-rev C x p = None⟧ ⟹
                           x ∉ dom (C r)
proof (induct p rule: rev-induct)
  case Nil show ?case using Nil
    by (simp add: applied-rule-rev-def)
next
  case (snoc y ys) show ?case using snoc
  proof (cases r ∈ set ys)
    case True show ?thesis using snoc True
```

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

2.6.2 Distributivity of the Transformation.

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

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

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

fun *prod-list* :: $('\alpha \mapsto '\beta) \Rightarrow (('\gamma \mapsto '\delta)$ *list*$) \Rightarrow$
$(('\alpha \mapsto '\beta) \Rightarrow ('\gamma \mapsto '\delta) \Rightarrow (('\alpha \times '\gamma) \mapsto ('\beta \times '\delta))) \Rightarrow$
$(('\beta \times '\delta) \Rightarrow 'y) \Rightarrow ('x \Rightarrow ('\alpha \times '\gamma)) \Rightarrow$
$(('x \mapsto 'y)$ *list*$)$ (**infixr** ‹$\bigotimes_L$› *54*) **where**
prod-list x (*y#ys*) *par-comb ran-adapt dom-adapt* =
((*ran-adapt o-f* ((*par-comb x y*) *o dom-adapt*))#(*prod-list x ys par-comb ran-adapt dom-adapt*))
| *prod-list x* [] *par-comb ran-adapt dom-adapt* = []

An instance, as usual there are four of them.

definition *prod-2-list* :: $[('\alpha \mapsto '\beta), (('\gamma \mapsto '\delta)$ *list*$)] \Rightarrow$
$(('\beta \times '\delta) \Rightarrow 'y) \Rightarrow ('x \Rightarrow ('\alpha \times '\gamma)) \Rightarrow$
$(('x \mapsto 'y)$ *list*$)$ (**infixr** ‹$\bigotimes_{2L}$› *55*) **where**
$x \bigotimes_{2L} y = (\lambda\ d\ r.\ (x \bigotimes_L y)\ (\bigotimes_2)\ d\ r)$

lemma *list2listNMT*: $x \neq [] \Longrightarrow$ *map sem* $x \neq []$
  apply (*case-tac x*)
   apply (*simp-all*)
  done

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

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

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

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

lemma *is-distr-orD*: $is\text{-}distr\ (\bigotimes_{\vee D})\ d\ r$
  apply (*simp add*: *is-distr-def*)
  apply (*rule allI*)+
  apply (*rule distr-orD*)
  apply (*simp*)
  done

lemma *is-strict-orD*: $is\text{-}strict\ (\bigotimes_{\vee D})\ d\ r$
  apply (*simp add*: *is-strict-def*)
  apply (*simp add*: *policy-range-comp-def*)
  done

lemma *is-distr-2*: $is\text{-}distr\ (\bigotimes_2)\ d\ r$
  apply (*simp add*: *is-distr-def*)
  apply (*rule allI*)+
  apply (*rule distr-or2*)
  by *simp*

lemma *is-strict-2*: $is\text{-}strict\ (\bigotimes_2)\ d\ r$
  apply (*simp only*: *is-strict-def*)
  apply *simp*
  apply (*simp add*: *policy-range-comp-def*)
  done

lemma *domStart*: $t \in dom\ p1 \Longrightarrow (p1 \bigoplus p2)\ t = p1\ t$
  apply (*simp add*: *map-add-dom-app-simps*)
  done

lemma *notDom*: $x \in dom\ A \Longrightarrow \neg\ A\ x = None$
  apply *auto*
  done

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

lemma *Norm-Distr-1*: $((r\ o\text{-}f\ (((\bigotimes_1)\ P1\ (list2policy\ P2))\ o\ d))\ x =$
$((list2policy\ ((P1 \bigotimes_L P2)\ (\bigotimes_1)\ r\ d))\ x))$

```
proof (induct P2)
  case Nil show ?case
    by (simp add: policy-range-comp-def  list2policy-def)
next
  case (Cons p ps) show ?case using Cons
  proof (cases x ∈ dom (r o-f ((P1 ⨂_1 p) ∘ d)))
    case True show ?thesis using True
      by (auto simp: list2policy-def policy-range-comp-def  prod-1-def
          split: option.splits decision.splits prod.splits)
  next
    case False show ?thesis using Cons False
       by (auto simp: list2policy-def policy-range-comp-def   map-add-dom-app-simps(3)
prod-1-def
          split: option.splits decision.splits prod.splits)
  qed
qed

lemma Norm-Distr-2: ((r o-f (((⨂_2) P1 (list2policy P2)) o d)) x =
                        ((list2policy ((P1 ⨂_L P2) (⨂_2) r d)) x))proof (induct P2)
  case Nil show ?case
    by (simp add: policy-range-comp-def  list2policy-def)
next
  case (Cons p ps) show ?case using Cons
  proof (cases x ∈ dom (r o-f ((P1 ⨂_2 p) ∘ d)))
    case True show ?thesis using True
      by (auto simp: list2policy-def prod-2-def policy-range-comp-def
          split: option.splits decision.splits prod.splits)
  next
    case False show ?thesis using Cons False
       by (auto simp:  policy-range-comp-def  list2policy-def map-add-dom-app-simps(3)
prod-2-def
          split: option.splits decision.splits prod.splits)
  qed
qed

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

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

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

Some domain reasoning

```
lemma domSubsetDistr1: dom A = UNIV ⟹ dom ((λ(x, y). x) o-f (A ⨂1 B) o (λ
x. (x,x))) = dom B
  apply (rule set-eqI)
  apply (rule iffI)
   apply (auto simp: prod-1-def policy-range-comp-def dom-def
      split: decision.splits option.splits prod.splits)
  done
```

```
lemma domSubsetDistr2: dom A = UNIV ⟹ dom ((λ(x, y). x) o-f (A ⨂2 B) o (λ
x. (x,x))) = dom B
  apply (rule set-eqI)
  apply (rule iffI)
   apply (auto simp: prod-2-def policy-range-comp-def dom-def
```

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

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

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

2.7 Policy Transformation for Testing

theory
  NormalisationTestSpecification
  imports
   Normalisation
begin

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

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

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

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

fun *PUTList* :: $('a \mapsto 'b) \Rightarrow 'a \Rightarrow ('a \mapsto 'b)\ list \Rightarrow bool$
 where
 $PUTList\ PUT\ x\ (p\#ps) = ((x \in dom\ p \longrightarrow (PUT\ x = p\ x)) \wedge (PUTList\ PUT\ x\ ps))$
$|PUTList\ PUT\ x\ [] = True$

lemma *distrPUTL1*: $x \in dom\ P \Longrightarrow (list2policy\ PL)\ x = P\ x$
$\Longrightarrow\ (PUTList\ PUT\ x\ PL \Longrightarrow (PUT\ x = P\ x))$
  apply (*induct PL*)
   apply (*auto simp*: *list2policy-def dom-def*)
  done

lemma *PUTList-None*: $x \notin dom\ (list2policy\ list) \Longrightarrow PUTList\ PUT\ x\ list$
  apply (*induct list*)
   apply (*auto simp*: *list2policy-def dom-def*)
  done

lemma *PUTList-DomMT*:
  $(\forall\, y{\in}set\ list.\ dom\ a \cap dom\ y = \{\}) \Longrightarrow x \in (dom\ a) \Longrightarrow x \notin dom\ (list2policy\ list)$
  apply (*induct list*)
   apply (*auto simp*: *dom-def list2policy-def*)
  done

lemma *distrPUTL2*:
  $x \in dom\ P \Longrightarrow (list2policy\ PL)\ x = P\ x\ \Longrightarrow disjDom\ PL \Longrightarrow (PUT\ x = P\ x) \Longrightarrow$
$PUTList\ PUT\ x\ PL$
  apply (*induct PL*)
   apply (*simp-all add*: *list2policy-def*)
  apply (*auto*)[*1*]
  subgoal for *a PL p*
    apply (*case-tac* $x \in dom\ a$)
     apply (*case-tac* $list2policy\ PL\ x = P\ x$)
      apply (*simp add*: *list2policy-def*)
     apply (*rule PUTList-None*)
     apply (*rule-tac* $a = a$ **in** *PUTList-DomMT*)
      apply (*simp-all add*: *list2policy-def dom-def*)
    done
  done

lemma *distrPUTL*:
  $[\![x \in dom\ P;\ (list2policy\ PL)\ x = P\ x;\ disjDom\ PL]\!] \Longrightarrow (PUT\ x = P\ x) = PUTList$
$PUT\ x\ PL$
  apply (*rule iffI*)
   apply (*rule distrPUTL2*)
      apply (*simp-all*)
  apply (*rule-tac* $PL = PL$ **in** *distrPUTL1*)
    apply (*auto*)
  done

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

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

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

definition *PUTListGD* **where** $PUTListGD\ PUT\ x\ p =$
  $(((x \notin (gatherDomain\ p) \wedge x \in dom\ (last\ p)) \longrightarrow PUT\ x = (last\ p)\ x) \wedge$
      $(PUTList\ PUT\ x\ (butlast\ p)))$

definition *disjDomGD* **where** $disjDomGD\ p = disjDom\ (butlast\ p)$

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

lemma *distrPUTLG2*:
  $PL \neq [] \Longrightarrow x \in dom\ P \Longrightarrow (list2policy\ (PL))\ x = P\ x \Longrightarrow disjDomGD\ PL \Longrightarrow$
   $(PUT\ x = P\ x) \Longrightarrow PUTListGD\ PUT\ x\ (PL)$
  apply (*simp add*: *PUTListGD-def disjDomGD-def gatherDomain-def list2policy-def*)
  apply (*induct PL*)
   apply (*auto*)
  apply (*metis PUTList-DomMT PUTList-None domI*)
done

lemma *distrPUTLG*:
  $[\![x \in dom\ P;\ (list2policy\ PL)\ x = P\ x;\ disjDomGD\ PL;\ PL \neq []]\!] \Longrightarrow$
  $(PUT\ x = P\ x) = PUTListGD\ PUT\ x\ PL$
  apply (*rule iffI*)
   apply (*rule distrPUTLG2*)
       apply (*simp-all*)
  apply (*rule-tac PL = PL* **in** *distrPUTLG1*)
    apply (*auto*)
  done

end

2.8 Putting Everything Together: UPF

theory
  UPF
  imports
    Normalisation
    NormalisationTestSpecification
    Analysis
begin

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

lemmas *UPFDefs* = *UPFCoreDefs ParallelDefs ElementaryPoliciesDefs*
end

3 Example

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

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

3.1 Secure Service Specification

theory
  Service
  imports
    UPF
begin

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

3.1.1 Datatypes for Modelling Users and Roles

Users

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

type-synonym *user* = *int*

Similarly, each patient has a unique id:

type-synonym *patient* = *int*

Roles and Relationships

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

datatype *role* = *ClinicalPractitioner* | *Nurse* | *Clerical*

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

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

The security context stores all the existing LRs.

type-synonym Σ = *patient* $\rightharpoonup$ *LR*

The user context stores the roles the users are in.

type-synonym υ = *user* $\rightharpoonup$ *role*

3.1.2 Modelling Health Records and the Web Service API

Health Records

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

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

The Web Service API

The operations provided by the service:

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

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

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

fun *is-deleteEntry* **where**

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

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

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

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

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

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

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

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

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

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

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

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

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

fun $contentStatic$ :: $Operation \Rightarrow bool$ 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
```

```
  $allContentStatic\ (x\#xs) = (contentStatic\ x \wedge allContentStatic\ xs)$
 $|allContentStatic\ [] = True$
```

3.1.3 Modelling Access Control

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

Sealed Envelopes

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

```
definition $get\text{-}entry :: DB \Rightarrow patient \Rightarrow entry\text{-}id \Rightarrow entry\ option$ where
 $get\text{-}entry\ S\ p\ e\text{-}id = \ \ (case\ S\ p\ of\ \bot \Rightarrow \bot$
                             $|\ \lfloor Scr \rfloor \Rightarrow Scr\ e\text{-}id)$

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

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

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

definition $editEntrySE :: SEPolicy$ where
 $editEntrySE\ x = (case\ x\ of\ (editEntry\ u\ r\ p\ e\text{-}id\ s,S) \Rightarrow \ (case\ get\text{-}entry\ S\ p\ e\text{-}id\ of$
                                                     $\bot \Rightarrow \bot$
                                                   $|\ \lfloor e \rfloor \Rightarrow (if\ (userHasAccess\ u\ e)$
                                                          $then\ \lfloor allow\ () \rfloor$
                                                          $else\ \lfloor deny\ () \rfloor))$
```

$| \ x \Rightarrow \bot)$

definition *SEPolicy* :: *SEPolicy* **where**
$SEPolicy = \ editEntrySE \bigoplus deleteEntrySE \bigoplus readEntrySE \bigoplus A_U$

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

Legitimate Relationships

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

fun $hasLR$:: $user \Rightarrow patient \Rightarrow \Sigma \Rightarrow bool$ **where**

$$\begin{aligned}
&hasLR\ u\ p\ \Sigma = (case\ \Sigma\ p\ of\ \bot \quad \Rightarrow False \\
&\qquad\qquad\qquad | \ \lfloor lrs \rfloor \ \Rightarrow \ (\exists\, lr.\ lr{\in}(ran\ lrs) \wedge u \in lr))
\end{aligned}$$

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

$$\begin{aligned}
&LRPolicy = (\lambda(x,y).\ (if\ hasLR\ (userOfOp\ x)\ (patientOfOp\ x)\ y \\
&\qquad\qquad then\ \lfloor allow\ ()\rfloor \\
&\qquad\qquad else\ \lfloor deny\ ()\rfloor))
\end{aligned}$$

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

$$\begin{aligned}
&createSCRPolicy\ x = (if\ \ (is\text{-}createSCR\ (fst\ x)) \\
&\qquad\qquad then\ \lfloor allow\ ()\rfloor \\
&\qquad\qquad else\ \ \bot)
\end{aligned}$$

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

$$\begin{aligned}
&addLRPolicy\ x = (if\ \ (is\text{-}addLR\ (fst\ x)) \\
&\qquad\qquad then\ \lfloor allow\ ()\rfloor \\
&\qquad\qquad else\ \bot)
\end{aligned}$$

definition *LR-Policy* **where** $LR\text{-}Policy = createSCRPolicy \bigoplus addLRPolicy \bigoplus LRPolicy \bigoplus A_U$

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

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

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

$$\begin{aligned}
&createFunPolicy\ ((createSCR\ u\ r\ p),(D,S)) = (if\ p \in dom\ D \\
&\qquad\qquad\qquad then\ \lfloor deny\ ()\rfloor \\
&\qquad\qquad\qquad else\ \lfloor allow\ ()\rfloor) \\
&|createFunPolicy\ x = \bot
\end{aligned}$$

```
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 = ⊥
```

```
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 ⨁ A_U
```

Modelling Core RBAC

type-synonym *RBACPolicy* = *Operation* × *υ* ↦ *unit*

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

```
    {(r,f). r = Clerical ∧ is-removeLR f}

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

3.1.4 The State Transitions and Output Function

State Transition

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


fun OpSuccessSigma :: (Operation × Σ) ⇀ Σ where
   OpSuccessSigma (addLR u r p lr-id us,S) =
                  (case S p of ⌊lrs⌋  ⇒ (case (lrs lr-id) of
                                           ⊥  ⇒ ⌊S(p↦(lrs(lr-id↦us)))⌋
                                         | ⌊x⌋ ⇒ ⌊S⌋)
                             | ⊥ ⇒ ⌊S(p↦(Map.empty(lr-id↦us)))⌋)
  |OpSuccessSigma (removeLR u r p lr-id,S) =
```

$(case\ S\ p\ of\ Some\ lrs \Rightarrow \lfloor S(p \mapsto (lrs(lr\text{-}id := \bot))) \rfloor$
$\quad | \bot \Rightarrow \lfloor S \rfloor)$
$|OpSuccessSigma\ (x,S) = \lfloor S \rfloor$

fun $OpSuccessUC :: (Operation \times \upsilon) \rightharpoonup \upsilon$ **where**
$OpSuccessUC\ (f,u) = \lfloor u \rfloor$

Output

type-synonym $Output = unit$

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

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

3.1.5 Combine All Parts

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

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

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

definition $ST\text{-}Allow :: Operation \times DB \times \Sigma\ \times \upsilon \rightharpoonup Output \times DB \times \Sigma \times \upsilon$
where $ST\text{-}Allow = ((OpSuccessOutput \bigotimes_M (OpSuccessDB \bigotimes_S OpSuccessSigma$
$\bigotimes_S OpSuccessUC))$
$\quad o\ (\ (\lambda(a,b,c).\ ((a),(a,b,c)))))$

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

definition $SE\text{-}LR\text{-}RBAC\text{-}ST\text{-}Policy :: Operation \times DB \times \Sigma\ \times \upsilon \mapsto Output \times DB \times$

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

definition *PolMon* :: *Operation* ⇒ (*Output decision*,*DB* × $\Sigma \times \upsilon$) MON_{SE}
where *PolMon* = (*policy2MON SE-LR-RBAC-ST-Policy*)

end

3.2 Instantiating Our Secure Service Example

theory
ServiceExample
imports
Service
begin

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

3.2.1 Access Control Configuration

definition *alice* :: *user* **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* :: υ **where**
UC0 = *Map.empty*(*alice*↦*Nurse*, *bob*↦*ClinicalPractitioner*, *charlie*↦*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*↦*entry1*))

definition *SCR2* :: *SCR* **where**
SCR2 = (*Map.empty*)

definition *Spine0* :: *DB* **where**
Spine0 = *Map.empty*(*patient1*↦*SCR1*, *patient2*↦*SCR2*)

definition *LR1* :: *LR* **where**
LR1 =(*Map.empty*(*1*↦{*alice*}))

definition Σ*0* :: Σ **where**
Σ*0* = (*Map.empty*(*patient1*↦*LR1*))

3.2.2 The Initial System State

definition σ*0* :: *DB* × Σ×υ **where**
σ*0* = (*Spine0*,Σ*0*,*UC0*)

3.2.3 Basic Properties

lemma [*simp*]: (*case a of allow d* ⇒ ⌊*X*⌋ | *deny d2* ⇒ ⌊*Y*⌋) = ⊥ ⟹ *False*
  by (*case-tac a*,*simp-all*)

lemma [*cong*,*simp*]:
((*if hasLR urp1-alice 1* Σ*0 then* ⌊*allow* ()⌋ *else* ⌊*deny* ()⌋) = ⊥) = *False*
  by (*simp*)

lemmas *MonSimps* = *valid-SE-def unit-SE-def bind-SE-def*
lemmas *Psplits* = *option.splits unit.splits prod.splits decision.splits*
lemmas *PolSimps* = *valid-SE-def unit-SE-def bind-SE-def if-splits policy2MON-def*
SE-LR-RBAC-ST-Policy-def map-add-def id-def LRsimps prod-2-def RBACPolicy-def
SE-LR-Policy-def SEPolicy-def RBAC-def deleteEntrySE-def editEntrySE-def
readEntrySE-def σ*0-def* Σ*0-def UC0-def patient1-def patient2-def LR1-def*
alice-def bob-def charlie-def get-entry-def SE-LR-RBAC-Policy-def Allow-def
Deny-def dom-restrict-def policy-range-comp-def prod-orA-def prod-orD-def
ST-Allow-def ST-Deny-def Spine0-def SCR1-def SCR2-def entry1-def entry2-def
entry3-def FunPolicy-def SE-LR-FUN-Policy-def o-def image-def UPFDefs

lemma *SE-LR-RBAC-Policy* ((*createSCR alice Clerical patient1*),$\sigma 0$)= *Some* (*deny* ())
  by (*simp add*: *PolSimps*)

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

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

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

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

lemma
  ($\sigma 0$ $\models$ (*os* $\leftarrow$ *mbind* [(*createSCR alice Clerical patient1*)] (*PolMon*);
    (*return* (*os* = [(*deny* (*Out*))]))))
  by (*simp add*: *PolMon-def MonSimps PolSimps*)

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

lemma *SE-LR-RBAC-Policy* ((*appendEntry alice Clerical patient1 ei d*),$\sigma 0$)= $\lfloor$*deny* ()$\rfloor$
  by (*simp add*: *PolSimps*)

end

4 Conclusion and Related Work

4.1 Related Work

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

4.2 Conclusion Future Work

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

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

5 Appendix

5.1 Basic Monad Theory for Sequential Computations

theory
  Monads
  imports
    Main
begin

5.1.1 General Framework for Monad-based Sequence-Test

As such, Higher-order Logic as a purely functional specification formalism has no built-in mechanism for state and state-transitions. Forms of testing involving state require therefore explicit mechanisms for their treatment inside the logic; a well-known technique to model states inside purely functional languages are *monads* made popular by Wadler and Moggi and extensively used in Haskell. HOLis 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 $('o,\ '\sigma)\ MON_{SE} = '\sigma \rightharpoonup ('o \times '\sigma)$

definition $\textit{bind-SE} :: ('o,'\sigma)MON_{SE} \Rightarrow ('o \Rightarrow ('o','\sigma)MON_{SE}) \Rightarrow ('o','\sigma)MON_{SE}$
where $\textit{bind-SE}\ f\ g = (\lambda\sigma.\ case\ f\ \sigma\ of\ None \Rightarrow None$
$\qquad\qquad\qquad\qquad\qquad\qquad | \ Some\ (out,\ \sigma') \Rightarrow g\ out\ \sigma')$

notation *bind-SE* (‹$bind_{SE}$›)
syntax

-bind-SE :: [*pttrn*,$('o,'\sigma)MON_{SE}$,$('o','\sigma)MON_{SE}$] ⇒ $('o','\sigma)MON_{SE}$
(‹(*2* - ← -; -)› [*5*,*8*,*8*]*8*)

syntax-consts
-bind-SE ⇌ *bind-SE*

translations
x ← *f*; *g* ⇌ *CONST bind-SE f* (% *x* . *g*)

definition *unit-SE* :: $'o \Rightarrow ('o, '\sigma)MON_{SE}$ (‹(*return* -)› *8*)
where *unit-SE e* = (λσ. *Some*(*e*,σ))
notation *unit-SE* (‹$unit_{SE}$›)

definition $fail_{SE}$ :: $('o, '\sigma)MON_{SE}$
where $fail_{SE}$ = (λσ. *None*)
notation $fail_{SE}$ (‹$fail_{SE}$›)

definition *assert-SE* :: ('σ ⇒ *bool*) ⇒ $(bool, '\sigma)MON_{SE}$
where *assert-SE P* = (λσ. *if P* σ *then Some*(*True*,σ) *else None*)
notation *assert-SE* (‹$assert_{SE}$›)

definition *assume-SE* :: ('σ ⇒ *bool*) ⇒ $(unit, '\sigma)MON_{SE}$
where *assume-SE P* = (λσ. *if* ∃σ . *P* σ *then Some*((), *SOME* σ . *P* σ) *else None*)
notation *assume-SE* (‹$assume_{SE}$›)

definition *if-SE* :: ['σ ⇒ *bool*, $('\alpha, '\sigma)MON_{SE}$, $('\alpha, '\sigma)MON_{SE}$] ⇒ $('\alpha, '\sigma)MON_{SE}$
where *if-SE c E F* = (λσ. *if c* σ *then E* σ *else F* σ)
notation *if-SE* (‹if_{SE}›)

The standard monad theorems about unit and associativity:

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

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

lemma *bind-assoc*: (*y* ← (*x* ← *m*; *k*); *h*) = (*x* ← *m*; (*y* ← *k*; *h*))
  apply (*simp add*: *unit-SE-def bind-SE-def*)
  apply (*rule ext*)

```
subgoal for σ
  apply (case-tac m σ, simp-all)
  subgoal for a
    apply (case-tac a, simp-all)
    done
  done
done
```

In order to express test-sequences also on the object-level and to make our theory amenable to formal reasoning over test-sequences, we represent them as lists of input and generalize the bind-operator of the state-exception monad accordingly. The approach is straightforward, but comes with a price: we have to encapsulate all input and output data into one type. Assume that we have a typed interface to a module with the operations op_1, op_2, ..., op_n with the inputs ι_1, ι_2, ..., ι_n (outputs are treated analogously). Then we can encode for this interface the general input - type:

$$\texttt{datatype in} = op_1 :: \iota_1 \mid ... \mid \iota_n$$

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

In order to express test-sequences also on the object-level and to make our theory amenable to formal reasoning over test-sequences, we represent them as lists of input and generalize the bind-operator of the state-exception monad accordingly. Thus, the notion of test-sequence is mapped to the notion of a *computation*, a semantic notion; at times we will use reifications of computations, i. e. a data-type in order to make computation amenable to case-splitting and meta-theoretic reasoning. To this end, we have to encapsulate all input and output data into one type. Assume that we have a typed interface to a module with the operations op_1, op_2, ..., op_n with the inputs ι_1, ι_2, ..., ι_n (outputs are treated analogously). Then we can encode for this interface the general input - type:

$$\texttt{datatype in} = op_1 :: \iota_1 \mid ... \mid \iota_n$$

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

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

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

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

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


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

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

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

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

definition *try-SE* :: $('o,'\sigma)$ $MON_{SE} \Rightarrow ('o\ option,'\sigma)\ MON_{SE}$
where *try-SE ioprog* = $(\lambda\sigma.\ case\ ioprog\ \sigma\ of$
$None \Rightarrow Some(None, \sigma)$
$|\ Some(outs, \sigma') \Rightarrow Some(Some\ outs, \sigma'))$

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

```
lemma mbind-try:
  (x ← mbind (a#S) F; M x) =
   (a' ← try-SE(F a);
     if a' = None
     then (M [])
     else (x ← mbind S F; M (the a' # x)))
  apply (rule ext)
  apply (simp add: bind-SE-def try-SE-def)
  subgoal for x
    apply (case-tac F a x)
     apply(simp)
    apply (safe)[1]
    apply (simp add: bind-SE-def try-SE-def)
    subgoal for aa b
      apply (case-tac mbind S F b)
       apply (auto)
      done
    done
  done
```

On this basis, a symbolic evaluation scheme can be established that reduces *mbind*-code to *try-SE*-code and If-cascades.

definition *alt-SE* :: $[('o, '\sigma)MON_{SE}, ('o, '\sigma)MON_{SE}] \Rightarrow ('o, '\sigma)MON_{SE}$ (**infixl** ‹$\sqcap_{SE}$› *10*)
where $(f \sqcap_{SE} g) = (\lambda\ \sigma.\ case\ f\ \sigma\ of\ None \Rightarrow g\ \sigma$
$|\ Some\ H \Rightarrow Some\ H)$

definition *malt-SE* :: $('o, '\sigma)MON_{SE}\ list \Rightarrow ('o, '\sigma)MON_{SE}$
where *malt-SE S* = *foldr alt-SE S* $fail_{SE}$
notation *malt-SE* (‹$\bigsqcap_{SE}$›)

lemma *malt-SE-mt* [*simp*]: $\bigsqcap_{SE}$ [] = $fail_{SE}$
  by(*simp add*: *malt-SE-def*)

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

State-Backtrack Monads

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

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

definition *bind-SB* :: $('o, '\sigma)MON_{SB} \Rightarrow ('o \Rightarrow ('o', '\sigma)MON_{SB}) \Rightarrow ('o', '\sigma)MON_{SB}$
where *bind-SB f g* $\sigma = \bigcup ((\lambda(out, \sigma).\ (g\ out\ \sigma))$ ‘ $(f\ \sigma))$
notation *bind-SB* (‹$bind_{SB}$›)

definition *unit-SB* :: $'o \Rightarrow ('o, '\sigma)MON_{SB}$ (‹(*returns* -)› *8*)
where *unit-SB e* $= (\lambda\sigma.\ \{(e,\sigma)\})$
notation *unit-SB* (‹$unit_{SB}$›)

syntax *-bind-SB* :: $[pttrn,('o,'\sigma)MON_{SB},('o','\sigma)MON_{SB}] \Rightarrow ('o','\sigma)MON_{SB}$
(‹(*2* - := -; -)› [*5*,*8*,*8*]*8*)
syntax-consts *-bind-SB* ⇌ *bind-SB*
translations
  x := *f*; *g* ⇌ *CONST bind-SB f* (% *x* . *g*)

lemma *bind-left-unit-SB* : (*x* := *returns a*; *m*) = *m*
  apply (*rule ext*)
  apply (*simp add*: *unit-SB-def bind-SB-def*)
  done

lemma *bind-right-unit-SB*: (*x* := *m*; *returns x*) = *m*
  apply (*rule ext*)
  apply (*simp add*: *unit-SB-def bind-SB-def*)
done

lemma *bind-assoc-SB*: (*y* := (*x* := *m*; *k*); *h*) = (*x* := *m*; (*y* := *k*; *h*))
  apply (*rule ext*)
  apply (*simp add*: *unit-SB-def bind-SB-def split-def*)
done

State Backtrack Exception Monad

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

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

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

definition *bind-SBE* $::$ $('o,'\sigma)MON_{SBE} \Rightarrow ('o \Rightarrow ('o','\sigma)MON_{SBE}) \Rightarrow ('o','\sigma)MON_{SBE}$
where *bind-SBE f g* $= (\lambda\sigma.\ case\ f\ \sigma\ of\ None \Rightarrow None$
$\qquad \mid Some\ S \Rightarrow (let\ S' = (\lambda(out,\ \sigma').\ g\ out\ \sigma')\ `\ S$
$\qquad\qquad in\ \ if\ None \in S'\ then\ None$
$\qquad\qquad\quad else\ Some(\bigcup\ (the\ `\ S'))))$

syntax *-bind-SBE* $:: [pttrn,('o,'\sigma)MON_{SBE},('o','\sigma)MON_{SBE}] \Rightarrow ('o','\sigma)MON_{SBE}$
$(\langle(2\ \text{-}\ :\equiv\ \text{-};\ \text{-})\rangle\ [5,8,8]8)$
syntax-consts *-bind-SBE* $\rightleftharpoons$ *bind-SBE*
translations
$\qquad x :\equiv f;\ g \rightleftharpoons CONST\ bind\text{-}SBE\ f\ (\%\ x\ .\ g)$

definition *unit-SBE* $::\ 'o \Rightarrow ('o,\ '\sigma)MON_{SBE}$ $(\langle(returning\ \text{-})\rangle\ 8)$
where *unit-SBE e* $= (\lambda\sigma.\ Some(\{(e,\sigma)\}))$

definition *assert-SBE* $::\ ('\sigma \Rightarrow bool) \Rightarrow (unit,\ '\sigma)MON_{SBE}$
where *assert-SBE e* $= (\lambda\sigma.\ if\ e\ \sigma\ then\ Some(\{((),\sigma)\})$
$\qquad else\ None)$
notation *assert-SBE* $(\langle assert_{SBE}\rangle)$

definition *assume-SBE* $::\ ('\sigma \Rightarrow bool) \Rightarrow (unit,\ '\sigma)MON_{SBE}$
where *assume-SBE e* $= (\lambda\sigma.\ if\ e\ \sigma\ then\ Some(\{((),\sigma)\})$
$\qquad else\ Some\ \{\})$
notation *assume-SBE* $(\langle assume_{SBE}\rangle)$

definition *havoc-SBE* $::\ (unit,\ '\sigma)MON_{SBE}$
where *havoc-SBE* $= (\lambda\sigma.\ Some(\{x.\ True\}))$
notation *havoc-SBE* $(\langle havoc_{SBE}\rangle)$

lemma *bind-left-unit-SBE* : $(x :\equiv returning\ a;\ m) = m$
  apply (*rule ext*)
  apply (*simp add*: *unit-SBE-def bind-SBE-def*)
  done

lemma *bind-right-unit-SBE*: $(x :\equiv m;\ returning\ x) = m$
  apply (*rule ext*)
  apply (*simp add*: *unit-SBE-def bind-SBE-def*)
  subgoal for x

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

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

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

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

qed

5.1.2 Valid Test Sequences in the State Exception Monad

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

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

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

lemma *syntax-test* :
  $\sigma \models (os \leftarrow (mbind\ \iota s\ ioprog);\ return(length\ \iota s = length\ os))$
oops

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

Recall mbind_unit for the base case.

lemma *valid-failure*: $ioprog\ a\ \sigma = None \Longrightarrow$
$(\sigma \models (s \leftarrow mbind\ (a\#S)\ ioprog\ ;\ M\ s)) =$
$(\sigma \models (M\ []))$
  by(*simp add*: *valid-SE-def unit-SE-def bind-SE-def*)

lemma *valid-failure'*: $A\ \sigma = None \Longrightarrow \neg(\sigma \models ((s \leftarrow A\ ;\ M\ s)))$
  by(*simp add*: *valid-SE-def unit-SE-def bind-SE-def*)

lemma *valid-successElem*:
$M\ \sigma = Some(f\ \sigma,\sigma) \Longrightarrow (\sigma \models M) = f\ \sigma$
  by(*simp add*: *valid-SE-def unit-SE-def bind-SE-def*)

lemma *valid-success*: $ioprog\ a\ \sigma = Some(b,\sigma') \Longrightarrow$
$(\sigma \models (s \leftarrow mbind\ (a\#S)\ ioprog\ ;\ M\ s)) =$
$(\sigma' \models (s \leftarrow mbind\ S\ ioprog\ ;\ M\ (b\#s)))$
  apply (*simp add*: *valid-SE-def unit-SE-def bind-SE-def*)
  apply (*cases mbind S ioprog* σ', *auto*)
  done

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

$(\sigma \models (s \leftarrow mbind\ (a\#S)\ ioprog\ ;\ return\ (P\ s))) =$
$(\sigma' \models (s \leftarrow mbind\ S\ ioprog\ ;\ return\ (P\ (b\#s))))$

apply (*simp add*: *valid-SE-def unit-SE-def bind-SE-def*)
apply (*cases mbind S ioprog σ'*)
 apply (*simp-all*)
apply (*auto*)
done

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

lemma *valid-both*: $(\sigma \models (s \leftarrow mbind\ (a\#S)\ ioprog\ ;\ return\ (P\ s))) =$
$(case\ ioprog\ a\ \sigma\ of$
$None \Rightarrow (\sigma \models (return\ (P\ [])))$
$\mid Some(b,\sigma') \Rightarrow (\sigma' \models (s \leftarrow mbind\ S\ ioprog\ ;\ return\ (P\ (b\#s)))))$

apply (*case-tac ioprog a σ*)
 apply (*simp-all add*: *valid-failure valid-success''* *split*: *prod.splits*)
done

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

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

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

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

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

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

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

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

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

lemma *assert-disch3* : $\neg\ P\ \sigma \Longrightarrow \neg\ (\sigma \models (assert_{SE}\ P))$
by(*auto simp*: *bind-SE-def assert-SE-def valid-SE-def*)

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

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

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

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

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

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

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

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

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

5.1.3 Valid Test Sequences in the State Exception Backtrack Monad

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

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

This notation considers all non-failures as valid.

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

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

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

end

Bibliography

[1] *American National Standard for Information Technology – Role Based Access Control.* ANSI, New York, Feb. 2004. ANSI INCITS 359-2004.

[2] C. A. Ardagna, S. D. C. di Vimercati, S. Foresti, T. W. Grandison, S. Jajodia, and P. Samarati. Access control for smarter healthcare using policy spaces. *Computers & Security*, 2010. ISSN 0167-4048. doi: 10.1016/j.cose.2010.07.001.

[3] S. Barker. The next 700 access control models or a unifying meta-model? In *Proceedings of the 14th ACM symposium on Access control models and technologies*, SACMAT '09, pages 187–196, New York, NY USA, 2009. ACM Press. ISBN 978-1-60558-537-6. doi: 10.1145/1542207.1542238.

[4] M. Y. Becker. Information governance in nhs's npfit: A case for policy specification. *International Journal of Medical Informatics*, 76(5-6):432–437, 2007. ISSN 1386-5056. doi: 10.1016/j.ijmedinf.2006.09.008.

[5] D. E. Bell. Looking back at the bell-la padula model. pages 337–351, Los Alamitos, CA, USA, 2005. pub-ieee. ISBN 1063-9527. doi: 10.1109/CSAC.2005.37.

[6] D. E. Bell and L. J. LaPadula. Secure computer systems: A mathematical model, volume II. In *Journal of Computer Security 4*, pages 229–263, 1996. An electronic reconstruction of *Secure Computer Systems: Mathematical Foundations*, 1973.

[7] E. Bertino, P. A. Bonatti, and E. Ferrari. Trbac: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001. ISSN 1094-9224. doi: 10.1145/501978.501979.

[8] A. D. Brucker and H. Petritsch. Extending access control models with break-glass. In B. Carminati and J. Joshi, editors, *ACM symposium on access control models and technologies (SACMAT)*, pages 197–206. ACM Press, New York, NY, USA, 2009. ISBN 978-1-60558-537-6. doi: 10.1145/1542207.1542239. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-extending-2009.

[9] A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 25(5):683–721, 2013. ISSN 0934-5043. doi: 10.1007/s00165-012-0222-y. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-theorem-prover-2012.

[10] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. An approach to modular and testable security models of real-world health-care applications. In

ACM symposium on access control models and technologies (SACMAT), pages 133–142, New York, NY, USA, 2011. ACM Press. ISBN 978-1-4503-0688-1. doi: 10.1145/1998441.1998461. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-model-based-2011.

[11] A. D. Brucker, L. Brügger, and B. Wolff. HOL-TestGen/FW: an environment for specification-based firewall conformance testing. In Z. Liu, J. Woodcock, and H. Zhu, editors, *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, number 8049 in Lecture Notes in Computer Science, pages 112–121. Springer-Verlag, Heidelberg, 2013. ISBN 978-3-642-39717-2. doi: 10.1007/978-3-642-39718-9_7. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-testgen-fw-2013.

[12] A. D. Brucker, L. Brügger, and B. Wolff. Formal firewall conformance testing: An application of test and proof techniques. *Software Testing, Verification & Reliability (STVR)*, 2014. doi: 10.1002/stvr.1544. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-formal-fw-testing-2014.

[13] L. Brügger. *A Framework for Modelling and Testing of Security Policies.* PhD thesis, ETH Zurich, 2012. URL http://www.brucker.ch/bibliography/abstract/bruegger-generation-2012. ETH Dissertation No. 20513.

[14] A. Ferreira, D. Chadwick, P. Farinha, G. Zhao, R. Chilro, R. Cruz-Correia, and L. Antunes. How to securely break into rbac: the btg-rbac model. In *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[15] N. Li, J. Byun, and E. Bertino. A critique of the ansi standard on role-based access control. *Security Privacy, IEEE*, 5(6):41–49, nov.-dec. 2007. ISSN 1540-7993. doi: 10.1109/MSP.2007.158.

[16] M. Moyer and M. Abamad. Generalized role-based access control. *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 391–398, Apr 2001. doi: 10.1109/ICDSC.2001.918969.

[17] OASIS. eXtensible Access Control Markup Language (XACML), version 2.0, 2005. URL http://docs.oasis-open.org/xacml/2.0/XACML-2.0-OS-NORMATIVE.zip.

[18] A. Samuel, A. Ghafoor, and E. Bertino. Context-aware adaptation of access-control policies. *Internet Computing, IEEE*, 12(1):51–54, 2008. ISSN 1089-7801. doi: 10.1109/MIC.2008.6.

[19] R. Sandhu, V. Bhamidipati, and Q. Munawer. The arbac97 model for role-based administration of roles. *ACM Transactions on Information and System Security*, 2 (1):105–135, 1999. ISSN 1094-9224. doi: 10.1145/300830.300839.

[20] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996. ISSN 0018-9162. URL http://ite.gmu.edu/list/journals/computer/pdf_ver/i94rbac(org).pdf.

[21] R. S. Sandhu, D. F. Ferraiolo, and D. R. Kuhn. The nist model for role-based access control: towards a unified standard. In *ACM Workshop on Role-Based Access Control*, pages 47–63, 2000. doi: 10.1145/344287.344301.

[22] J. Wainer, A. Kumar, and P. Barthelmess. Dw-rbac: A formal security model of delegation and revocation in workflow systems. *Inf. Syst.*, 32(3):365–384, 2007. ISSN 0306-4379. doi: https://doi.org/10.1016/j.is.2005.11.008.