The Unified Policy Framework

Achim D. Brucker Lukas Brügger Burkhart Wolff

October 15, 2012

Contents

1 The Unified Policy Framework (UPF)

theory *UPFCore*
imports *Main Testing*
begin

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. 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). From the modelling perspective, using HOL as its input language, we will consequently use the expressive power of its underlying functional programming language, including the possibility to define higher-order combinators.

In more detail, we model policies as partial functions based on input data 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*)
translations (*type*) $'\alpha$ |−> $'\beta$ <= (*type*) $'\alpha$ $\rightharpoonup$ $'\beta$ *decision*

type-notation (*xsymbols*)
policy (**infixr** $\mapsto$ *0*)

notation
None ($\bot$)

notation
Some ($\lfloor$-$\rfloor$ *80*)

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

We define the two fundamental sets, the allow-set *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*

1.2 Policy Constructors

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

However, we add notation on top of this 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)

syntax (*xsymbols*)
-policylet1 :: ['a, 'a] => *policylet* (- /+↦/ -)
-policylet2 :: ['a, 'a] => *policylet* (- /−↦/ -)
-emptypolicy :: 'a |-> 'b (∅)

translations
-MapUpd m (*-Maplets xy ms*) == *-MapUpd* (*-MapUpd m xy*) *ms*
-MapUpd m (*-policylet1 x y*) == *m*(*x* := *CONST Some* (*CONST allow y*))
-MapUpd m (*-policylet2 x y*) == *m*(*x* := *CONST Some* (*CONST deny y*))
∅ == *CONST empty*

lemma *test*: *empty*(*x*+=*a*, *y*−= *b*) = ∅(*x* +↦ *a*, *y* −↦ *b*) **by** *simp*

lemma *test2*: *p*(*x*+↦*a*,*x*−↦*b*) = *p*(*x*−↦ *b*) **by** *simp*

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

lemma *pol-upd-triv1*: *t k* = ⌊*allow x*⌋ ⟹ *t*(*k*+↦*x*) = *t*
by (*rule ext*) *simp*
lemma *pol-upd-triv2*: *t k* = ⌊*deny x*⌋ ⟹ *t*(*k*−↦*x*) = *t*
by (*rule ext*) *simp*

lemma *pol-upd-allow-nonempty*: *t*(*k*+↦*x*) ≠ ∅ **by** *simp*
lemma *pol-upd-deny-nonempty*: *t*(*k*−↦*x*) ≠ ∅ **by** *simp*

lemma *pol-upd-eqD1* : *m*(*a*+↦*x*) = *n*(*a*+↦*y*) ⟹ *x* = *y*
by(*auto dest*: *map-upd-eqD1*)

lemma *pol-upd-eqD2* : *m*(*a*−↦*x*) = *n*(*a*−↦*y*) ⟹ *x* = *y*
by(*auto dest*: *map-upd-eqD1*)

lemma *pol-upd-neq1* [*simp*]: *m*(*a*+↦*x*) ≠ *n*(*a*−↦*y*)
by(*auto dest*: *map-upd-eqD1*)

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** $(+p/)$ *100*)

syntax (*xsymbols*)
-policyoverride :: $['a \mapsto 'b, 'a \mapsto 'b] \Rightarrow 'a \mapsto 'b$ (**infixl** $\bigoplus$ *100*)

translations
$p \bigoplus q == q +\!\!+ p$

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* ::$['\alpha \mapsto '\beta, '\alpha \mapsto '\beta] \Rightarrow '\alpha \mapsto '\beta$ (**infixl** $+\!\!+'$*-A 100*)
where *m2* $+\!\!+$*-A m1* =
$(\lambda x.$ (*case m1 x of*
$\lfloor$*allow a*$\rfloor \Rightarrow \lfloor$*allow a*$\rfloor$
| $\lfloor$*deny a*$\rfloor \Rightarrow$ (*case m2 x of* $\lfloor$*allow b*$\rfloor \Rightarrow \lfloor$*allow b*$\rfloor$
| - $\Rightarrow \lfloor$*deny a*$\rfloor$)
| $\bot \Rightarrow$ *m2 x*)
)

syntax (*xsymbols*)
-policyoverride-A :: $['a \mapsto 'b, 'a \mapsto 'b] \Rightarrow 'a \mapsto 'b$ (**infixl** $\bigoplus_A$ *100*)

translations
$p \bigoplus_A q == p +\!\!+$*-A* q

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

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

lemma *override-A-assoc*:

$p1 \bigoplus_A (p2 \bigoplus_A p3) = (p1 \bigoplus_A p2) \bigoplus_A p3$
by (*rule ext*, *simp add*: *override-A-def split*: *decision.splits option.splits*)

definition *override-D* :: [$'\alpha \mapsto 'b$, $'\alpha \mapsto 'b$] ⇒ $'\alpha \mapsto 'b$ (**infixl** ++$'$-*D 100*)
where *m1 ++-D m2* =
(λ*x*. *case m2 x of*
⌊*deny a*⌋ ⇒ ⌊*deny a*⌋
| ⌊*allow a*⌋ ⇒ (*case m1 x of* ⌊*deny b*⌋ ⇒ ⌊*deny b*⌋
| - ⇒ ⌊*allow a*⌋)
| ⊥ ⇒ *m1 x*
)

syntax (*xsymbols*)
-policyoverride-D :: [$'a \mapsto 'b$, $'a \mapsto 'b$] ⇒ $'a \mapsto 'b$ (**infixl** $\bigoplus_D$ *100*)
translations
$p \bigoplus_D q$ == *p ++-D q*

lemma *override-D-empty*[*simp*]: $p \bigoplus_D \emptyset = p$
by(*simp add*:*override-D-def*)

lemma *empty-override-D*[*simp*]: $\emptyset \bigoplus_D p = p$
apply(*rule ext*, *simp add*:*override-D-def*)
apply(*case-tac p x*, *simp-all*)
apply(*case-tac a*, *simp-all*)
done

lemma *override-D-assoc*:
$p1 \bigoplus_D (p2 \bigoplus_D p3) = (p1 \bigoplus_D p2) \bigoplus_D p3$
by (*rule ext*, *simp add*: *override-D-def split*: *decision.splits option.splits*)

1.4 Coercion Operators

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

The domain of a policy can be adapted via a conventional function composition:

lemma *test* : $(p::'\beta \mapsto '\gamma)\ o\ (f::'\alpha \Rightarrow '\beta) = (C::'\alpha \mapsto '\gamma)$
oops

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

definition *policy-range-comp* :: [$'\beta \Rightarrow '\gamma$, $'\alpha \mapsto '\beta$] ⇒ $'\alpha \mapsto '\gamma$
(**infixl** *o*$'$-*f 55*)
where
f o-f p = (λ*x*. *case p x of*
⌊*allow y*⌋ ⇒ ⌊*allow* (*f y*)⌋
| ⌊*deny y*⌋ ⇒ ⌊*deny* (*f y*)⌋
| ⊥ ⇒ ⊥)

syntax (*xsymbols*)

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

translations
$p\ o_f\ q == p$ *o-f* q

lemma *policy-range-comp-strict* : $f\ o_f\ \emptyset = \emptyset$
by(*rule ext*, *simp add*: *policy-range-comp-def*)

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

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

lemma *range-split-strict*[*simp*]: $P\ \nabla\ \emptyset = \emptyset$
by(*rule ext*, *simp add*: *range-split-def*)

lemma *range-split-charn*:
$(f,g)\ \nabla\ p = (\lambda x.$ *case* $p\ x$ *of*
$\lfloor$*allow* $x\rfloor \Rightarrow \lfloor$*allow* $(f\ x)\rfloor$
$\mid \lfloor$*deny* $x\rfloor \Rightarrow \lfloor$*deny* $(g\ x)\rfloor$
$\mid \bot \Rightarrow \bot)$
apply(*simp add*: *range-split-def*, *rule ext*)
apply(*case-tac* $p\ x$, *simp-all*)
apply(*case-tac* a, *simp-all*)
done

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

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

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

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

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

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

definition *dom-split2a* :: $[('\alpha \rightharpoonup '\gamma) \times ('\alpha \rightharpoonup '\gamma), '\alpha \mapsto '\beta] \Rightarrow '\alpha \mapsto '\gamma$
(**infixr** Δa *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** $\nabla 2$ *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)$\rfloor$
$\mid \bot \Rightarrow \bot)$

A common case: only allow results are propagated:

lemma (*Some*,$K\ \bot$) $\nabla\ p = (f::'a \mapsto 'b\ option)$
oops

Another common case considers transition policies: only allow results are propagated, however, in both cases, the state is propagated.

lemma $(\lambda(\iota,\sigma).$(*Some* ι,σ),$\lambda(\iota,\sigma).(\bot, \sigma))\ \nabla\ p = (f::'\alpha \times '\sigma \mapsto '\beta\ option \times '\sigma)$
oops

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.

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 (ι,σ) *of*
$\lfloor$(*allow* (*outs*,σ'))$\rfloor \Rightarrow \lfloor$(*allow outs*, σ')$\rfloor$
$\mid \lfloor$(*deny* (*outs*,σ'))$\rfloor \Rightarrow \lfloor$(*deny outs*, σ')$\rfloor$
$\mid \bot \Rightarrow \bot)$

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 Elementary Policies

theory *ElementaryPolicies*
imports *UPFCore*
begin

Injections and Bijections of Policies

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

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

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

translations
 $A_p\ f$ == *AllA f*

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

translations
 $D_p\ f$ == *AllD f*

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

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

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

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

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

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

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

lemma *neq-Allow-Deny*: *pf* ≠ ∅ ⟹ (*deny-pfun pf*) ≠ (*allow-pfun pf*)
apply(*erule contrapos-nn*, *rule ext*)
apply(*drule-tac x*=*x* **in** *fun-cong*)
apply(*auto simp*: *deny-pfun-def allow-pfun-def*)
apply(*case-tac pf x* = ⊥, *auto*)
done

Common instances :

definition *allow-all-fun* :: ('*α* ⇒ '*β*) ⇒ ('*α* ↦ '*β*) (*A*$_f$)
where *allow-all-fun f* = *allow-pfun* (*Some o f*)

definition *deny-all-fun* :: ('*α* ⇒ '*β*) ⇒ ('*α* ↦ '*β*) (*D*$_f$)
where *deny-all-fun f* ≡ *deny-pfun* (*Some o f*)

definition
deny-all-id :: '*α* ↦ '*α* (*D*$_I$) **where**
deny-all-id ≡ *deny-pfun* (*id o Some*)

definition
allow-all-id :: '*α* ↦ '*α* (*A*$_I$) **where**
allow-all-id ≡ *allow-pfun* (*id o Some*)

definition
allow-all :: ('*α* ↦ *unit*) (*A*$_U$) **where**

allow-all p $=$ *Some* (*allow* ())

definition
deny-all :: ($'\alpha \mapsto$ *unit*) (D_U) **where**
deny-all p $=$ *Some* (*deny* ())

... and resulting properties:

lemma $A_I \bigoplus$ *empty* $= A_I$
apply *simp*
done

lemma $A_f\ f \bigoplus$ *empty* $= A_f\ f$
apply *simp*
done

lemma *allow-pfun empty* $=$ *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*)+
apply (*auto simp*: *deny-pfun-def option.splits*)
done

3 Domain, Range, and Restictions

find-theorems *name*:*range*

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

However, some properties are specific to policy concepts:

lemma *sub-ran* : $ran\ p \subseteq Allow \cup Deny$
apply(*auto simp*: *Allow-def Deny-def ran-def full-SetCompr-eq*[*symmetric*])
apply(*case-tac x*, *simp-all*)
apply(*erule-tac x*=α **in** *allE*, *simp*)
done

lemma *dom-allow-pfun* [*simp*]:$dom(allow\text{-}pfun\ f) = dom\ f$
apply(*auto simp*: *allow-pfun-def*)
apply(*case-tac f x*, *simp-all*)
done

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

lemma *dom-deny-pfun* [*simp*]:$dom(deny\text{-}pfun\ f) = dom\ f$
apply(*auto simp*: *deny-pfun-def*)
apply(*case-tac f x*, *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*]:$ran(allow\text{-}pfun\ f) = allow\ `(ran\ f)$
apply(*simp add*: *allow-pfun-def ran-def*)
apply(*rule set-eqI*, *auto*)
apply(*case-tac f a*, *auto simp*: *image-def*)
apply(*rule-tac x*=*a* **in** *exI*, *simp*)
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*, *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*, *auto*)
apply(*case-tac f a*, *auto simp*: *image-def*)
apply(*rule-tac x*=*a* **in** *exI*, *simp*)
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*, *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* ∪ *dom ?m*
- `Map.inj_on_map_add_dom` *inj-on* (*?m′* $\bigoplus$ *?m*) (*dom ?m′*) = *inj-on ?m′* (*dom ?m′*)
- `Map.map_add_comm` *dom ?m1.0* ∩ *dom ?m2.0* = {} ⟹ *?m2.0* $\bigoplus$ *?m1.0* = *?m1.0* $\bigoplus$ *?m2.0*
- `Map.map_add_dom_app_simps(1)` *?m* ∈ *dom ?l2.0* ⟹ (*?l2.0* $\bigoplus$ *?l1.0*) *?m* = *?l2.0 ?m*
- `Map.map_add_dom_app_simps(2)` *?m* ∉ *dom ?l1.0* ⟹ (*?l2.0* $\bigoplus$ *?l1.0*) *?m* = *?l2.0 ?m*
- `Map.map_add_dom_app_simps(3)` *?m* ∉ *dom ?l2.0* ⟹ (*?l2.0* $\bigoplus$ *?l1.0*) *?m* = *?l1.0 ?m*
- `Map.map_add_upd_left` *?m* ∉ *dom ?e2.0* ⟹ *?e2.0* $\bigoplus$ *?e1.0*(*?m* ↦ *?u1.0*) = (*?e2.0* $\bigoplus$ *?e1.0*)(*?m* ↦ *?u1.0*)

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

definition *dom-restrict* :: [′α *set*, ′α↦′β] ⇒ ′α↦′β (**infixr** ◁ *55*)
where *S* ◁ *p* ≡ (λ*x*. *if x* ∈ *S then p x else* ⊥)

lemma *dom-dom-restrict*[*simp*] : *dom*(*S* ◁ *p*) = *S* ∩ *dom p*
apply(*auto simp*: *dom-restrict-def*)
apply(*case-tac x* ∈ *S*, *simp-all*)
apply(*case-tac x* ∈ *S*, *simp-all*)
done

lemma *dom-restrict-idem*[*simp*] : (*dom p*) ◁ *p* = *p*
by(*rule ext*,
  auto simp: *dom-restrict-def*
    dest: *neq-commute*[*THEN iffD1*,*THEN not-None-eq* [*THEN iffD1*]])

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

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

lemma *ran-ran-restrict*[*simp*] : $ran(p \rhd S) = S \cap ran\ p$
by(*auto simp*: *ran-restrict-def image-def ran-def*)

lemma *ran-restrict-idem*[*simp*] : $p \rhd (ran\ p) = p$
apply(*rule ext*, *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*] : $(p \rhd S) \rhd T = p \rhd T \cap S$
by(*rule ext*,
  auto simp: *ran-restrict-def*
    dest: *neq-commute*[*THEN iffD1*,*THEN not-None-eq* [*THEN iffD1*]])

lemma *ran-gen-A*[*simp*] : $(\forall Ax.\ \lfloor P\ x \rfloor) \rhd Allow = (\forall Ax.\ \lfloor P\ x \rfloor)$
by(*rule ext*, *auto simp*: *Allow-def ran-restrict-def*)

lemma *ran-gen-D*[*simp*] : $(\forall Dx.\ \lfloor P\ x \rfloor) \rhd Deny = (\forall Dx.\ \lfloor P\ x \rfloor)$
by(*rule ext*, *auto simp*: *Deny-def ran-restrict-def*)

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

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.

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* ::['α↦'β, 'γ ↦'δ] ⇒ ('α×'γ ↦ 'β×'δ)

(**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*, *simp add*: *prod-orA-def*)
apply (*auto*, *simp split*: *option.splits decision.splits*)
done

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

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*, *simp add*: *prod-orA-def policy-range-comp-def o-def*)
apply(*auto* ,*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*, *simp add*: *prod-orD-def*)
apply (*auto*, *simp split*: *option.splits decision.splits*)
done

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

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*, *simp add*: *prod-orD-def policy-range-comp-def o-def*)
apply (*auto*, *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*, *simp add*: *prod-1-def*)
apply (*auto*, *simp split*: *option.splits decision.splits*)
done

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

definition *prod-2* :: [$'\alpha \mapsto '\beta$, $'\gamma \mapsto '\delta$] $\Rightarrow$ ($'\alpha \times '\gamma \mapsto '\beta \times '\delta$)
(**infixr** $\bigotimes_2$ *55*)
where *p1* $\bigotimes_2$ *p2* $\equiv$
(λ(*x*,*y*). (*case p1 x of*
$\lfloor$*allow d1*$\rfloor$ $\Rightarrow$(*case p2 y of*
$\lfloor$*allow d2*$\rfloor$ $\Rightarrow$ $\lfloor$*allow*(*d1*,*d2*)$\rfloor$
| $\lfloor$*deny d2*$\rfloor$ $\Rightarrow$ $\lfloor$*deny* (*d1*,*d2*)$\rfloor$
| $\bot$ $\Rightarrow$ $\bot$)
| $\lfloor$*deny d1*$\rfloor$$\Rightarrow$(*case p2 y of*
$\lfloor$*allow d2*$\rfloor$ $\Rightarrow$ $\lfloor$*allow*(*d1*,*d2*)$\rfloor$
| $\lfloor$*deny d2*$\rfloor$ $\Rightarrow$ $\lfloor$*deny* (*d1*,*d2*)$\rfloor$
| $\bot$ $\Rightarrow$ $\bot$)
|$\bot$ $\Rightarrow$$\bot$))

lemma *prod-2-mt*[*simp*]:*p* $\bigotimes_2$ $\emptyset$ = $\emptyset$
apply(*rule ext*, *simp add*: *prod-2-def*)
apply (*auto*, *simp split*: *option.splits decision.splits*)
done

lemma *mt-prod-2*[*simp*]:$\emptyset \bigotimes_2 p = \emptyset$
by(*rule ext*, *simp add*: *prod-2-def*)

definition *prod-1-id* ::[$'\alpha \mapsto '\beta$, $'\alpha \mapsto '\gamma$] $\Rightarrow$ ($'\alpha \mapsto '\beta \times '\gamma$) (**infixr** $\bigotimes_{1I}$ *55*)
where *p* $\bigotimes_{1I}$ *q* = (*p* $\bigotimes_1$ *q*) *o* ($\lambda x.$ (*x*,*x*))

lemma *prod-1-id-mt*[*simp*]:*p* $\bigotimes_{1I}$ $\emptyset$ = $\emptyset$
apply(*rule ext*, *simp add*: *prod-1-id-def*)
done

lemma *mt-prod-1-id*[*simp*]:$\emptyset \bigotimes_{1I} p = \emptyset$
by(*rule ext*, *simp add*: *prod-1-id-def prod-1-def*)

definition *prod-2-id* ::[$'\alpha \mapsto '\beta$, $'\alpha \mapsto '\gamma$] $\Rightarrow$ ($'\alpha \mapsto '\beta \times '\gamma$) (**infixr** $\bigotimes_{2I}$ *55*)
where*p* $\bigotimes_{2I}$ *q* = (*p* $\bigotimes_2$ *q*) *o* ($\lambda x.$ (*x*,*x*))

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

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

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* = (λ (*x*,*y*). *case p1 x of* $\lfloor$*d1*$\rfloor$ $\Rightarrow$
(*case p2 y of* $\lfloor$*d2*$\rfloor$ $\Rightarrow$ $\lfloor$(*d1*,*d2*)$\rfloor$
| $\bot \Rightarrow \bot$)
| $\bot \Rightarrow \bot$)

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* (λ (*a*,*b*,*c*). ((*a*,*b*),*a*,*c*))

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*)
apply (*case-tac f x*)
apply (*simp-all add*: *prod-1-def map-add-def*
split: *decision.splits option.splits*)
done

lemma *distr-or1*: (*F* = *F1* $\bigoplus$ *F2*) $\Longrightarrow$ ((*g o-f* ((*N* $\bigotimes_1$ *F*) *o f*)) =
((*g o-f* ((*N* $\bigotimes_1$ *F1*) *o f*)) $\bigoplus$ (*g o-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*)
apply (*case-tac f x*)
apply (*simp-all add*: *prod-1-def map-add-def*
split: *decision.splits option.splits*)
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*)
apply (*case-tac f x*)
apply (*simp-all add*: *prod-2-def map-add-def*

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

lemma *distr-or2*: $(F = F1 \bigoplus F2) \Longrightarrow ((r\ o\text{-}f\ ((N \bigotimes_2 F)\ o\ f)) =$
$((r\ o\text{-}f\ ((N \bigotimes_2 F1)\ o\ f))\ \bigoplus\ (r\ o\text{-}f\ ((N \bigotimes_2 F2)\ o\ f))))$
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*)
apply (*case-tac f x*)
apply (*simp-all add*: *prod-2-def map-add-def*
split: *decision.splits option.splits*)
done

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

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

lemma *coerc-assoc*: $(r\ o\text{-}f\ P)\ o\ d = r\ o\text{-}f\ (P\ o\ d)$
apply (*simp add*: *policy-range-comp-def*)
apply (*rule ext*)
apply (*simp split*: *option.splits decision.splits*)
done

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** $\bigotimes_\triangledown$ *100*)
where $P \bigotimes_\triangledown p \equiv \lambda x.$ *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*:
$(f,\ g)\ \bigotimes_{\triangledown}\ p = ($
$(((p\ \triangleright\ Allow) \bigotimes_{\vee A}\ (A_p\ f))\ \bigoplus$
$((p\ \triangleright\ Deny)\ \bigotimes_{\vee A}\ (D_p\ g))))$
apply(*rule ext*)
by(*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*) *auto*

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

5 Sequential Composition

theory *SeqComposition*
imports *ParallelComposition*
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.

A key concept of sequential policy composition is the flattening of nested decisions. There are 4 possibilities, and these possibilities will give the various flavours of policy composition.

fun *flat-orA* :: ('α *decision*) *decision* $\Rightarrow$ ('α *decision*)
where *flat-orA*(*allow*(*allow y*)) = *allow y*
$\mid$*flat-orA*(*allow*(*deny y*)) = *allow y*
$\mid$*flat-orA*(*deny*(*allow y*)) = *allow y*
$\mid$*flat-orA*(*deny*(*deny y*)) = *deny y*

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

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

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

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

fun *flat-1* :: ('α *decision*) *decision* ⇒ ('α *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, *simp-all*)
apply(*case-tac* α, *simp-all*)
apply(*case-tac* α, *simp-all*)
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, *simp-all*)
apply(*case-tac* α, *simp-all*)
apply(*case-tac* α, *simp-all*)
done

fun *flat-2* :: (′α *decision*) *decision* ⇒ (′α *decision*)
where *flat-2*(*allow*(*allow y*)) = *allow y*
| *flat-2*(*allow*(*deny y*)) = *deny y*
| *flat-2*(*deny*(*allow y*)) = *allow y*
| *flat-2*(*deny*(*deny y*)) = *deny y*

lemma *flat-2-allow*[*dest*]:
flat-2 x = *allow y* ⟹ *x* = *allow*(*allow y*) ∨ *x* = *deny*(*allow y*)
apply(*case-tac x*, *simp-all*)
apply(*case-tac α*, *simp-all*)
apply(*case-tac α*, *simp-all*)
done

lemma *flat-2-deny*[*dest*]:
flat-2 x = *deny y* ⟹ *x* = *deny*(*deny y*) ∨ *x* = *allow*(*deny y*)
apply(*case-tac x*, *simp-all*)
apply(*case-tac α*, *simp-all*)
apply(*case-tac α*, *simp-all*)
done

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

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

lemma *lift-mt* [*simp*]: *lift* ∅ = ∅
by(*rule ext*, *case-tac x*,*simp-all*)

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 4 different forms of flattening, we have four different forms of policy composition:

definition
comp-orA :: [′β↦′γ, ′α↦′β] ⇒ ′α↦′γ (**infixl** *o′-orA 55*) **where**
p2 o-orA p1 ≡ (*Option.map flat-orA*) *o* (*lift p2 o-m p1*)

notation (*xsymbols*)
comp-orA (**infixl** ∘$_{\vee A}$ *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*)

lemma *comp-orA-assoc*:
p3 o-orA (*p2 o-orA p1*) =
p3 o-orA p2 o-orA p1
oops

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

notation (*xsymbols*)
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*)

lemma *comp-orD-assoc*:
p3 o-orD (*p2 o-orD p1*) = *p3 o-orD p2 o-orD p1*
apply(*simp add*: *comp-orD-def*)
oops

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

notation (*xsymbols*)
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*)

lemma *comp-1-assoc*:
$p3 \circ_1 (p2 \circ_1 p1) = p3 \circ_1 p2 \circ_1 p1$
apply(*simp add*: *comp-1-def*)

oops

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

notation (*xsymbols*)
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*)

lemma *comp-2-assoc*:
$p3 \circ_2 (p2 \circ_2 p1) = p3 \circ_2 p2 \circ_2 p1$
apply(*simp add*: *comp-2-def*)
oops

end

6 Algebraic Properties of Policies.

theory *Analysis*
imports
ParallelComposition
SeqComposition
begin

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

A policy has no gaps

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

Policy p is more defined than q

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

p is more permissive than q

definition *more-permissive* :: $('a \mapsto 'b) \Rightarrow ('a \mapsto 'b) \Rightarrow bool$ **where**
more-permissive p q = (∀ *x*. (*case q x of* ⌊*allow y*⌋ ⇒ (∃ *z*. (*p x* = ⌊*allow z*⌋))
| ⌊*deny y*⌋ ⇒ *True*
| ⊥ ⇒ *True*))

definition *more-rejective* :: $('a \mapsto 'b) \Rightarrow ('a \mapsto 'b) \Rightarrow bool$ **where**
more-rejective p q = $(\forall\ x.\ (case\ q\ x\ of\ \lfloor deny\ y \rfloor \Rightarrow (\exists\ z.\ (p\ x = \lfloor deny\ z \rfloor))$
$\mid \lfloor allow\ y \rfloor \Rightarrow True$
$\mid \bot \Rightarrow True))$

lemma *more-permissive* $(A_f\ f)\ p$
apply (*simp add*: *more-permissive-def allow-all-fun-def allow-pfun-def*)
apply (*rule allI*)
apply (*case-tac p x* ,*simp-all*)
apply (*case-tac a*,*simp-all*)
done

lemma *more-permissive* $A_I\ p$
apply (*simp add*: *more-permissive-def allow-all-fun-def allow-pfun-def allow-all-id-def*)
apply (*rule allI*)
apply (*case-tac p x* ,*simp-all*)
apply (*case-tac a*,*simp-all*)
done

Equivalence over domain D

definition *p-eq-dom* :: $('a \mapsto 'b) \Rightarrow ('a \mapsto 'b) \Rightarrow 'a\ set \Rightarrow bool$ **where**
p-eq-dom p q d = $(\forall x \in d.\ p\ x = q\ x)$

p and q have no conflicts

definition *no-conflicts* :: $('a \mapsto 'b) \Rightarrow ('a \mapsto 'b) \Rightarrow bool$ **where**
no-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*: *more-permissive p q*
and *q-over-pA*: *more-permissive q p*
and *p-over-qD*: *more-rejective q p*
and *q-over-pD*: *more-rejective p 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

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

by *auto*

lemma *dom-eq*: *dom p* ∩ *dom q* = {} ⟹
p ⊕$_A$ *q* = *p* ⊕$_D$ *q*
apply (*simp add*: *override-A-def override-D-def*)
apply (*rule ext*)
apply (*simp split*: *prod.splits option.splits decision.splits*)
apply (*simp add*: *dom-def*)
apply *auto*
done

lemma *dom-split-alt-def* :
(*f*, *g*) Δ *p* = (*dom*(*p* ▷ *Allow*) ◁ (*A*$_f$ *f*)) ⊕
(*dom*(*p* ▷ *Deny*) ◁ (*D*$_f$ *g*))
apply(*rule ext*)
apply(*simp add*: *dom-split2-def Allow-def Deny-def dom-restrict-def*
deny-all-fun-def allow-all-fun-def map-add-def)
apply(*simp split*:*option.splits decision.splits*)
apply(*auto simp*:*map-add-def o-def deny-pfun-def ran-restrict-def image-def*)
done

end

7 Policy Transformations

theory *Normalisation*
imports *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.

7.1 Elementary Operators

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

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

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

Determine the position of element of a list.

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

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

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

The following is used if the list is constructed backwards.

definition *applied-rule-rev* **where**
applied-rule-rev C a x = (*applied-rule C a* (*rev x*))

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

fun *removeShadowRules3* **where**
removeShadowRules3 C (*x*#*xs*) = (*if* (*dom* (*C x*)= {}) *then* (*removeShadowRules3 C xs*)
else (*x*#(*removeShadowRules3 C xs*)))
|*removeShadowRules3 C* [] = []

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

fun *noneMT* **where**
noneMT C (*x*#*xs*) = (*dom* (*C x*) ≠ {} ∧ (*noneMT C xs*))
|*noneMT C* [] = *True*

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

fun *notMTpolicy* **where**
notMTpolicy C (*x*#*xs*) = (*if* (*dom* (*C x*) = {}) *then* (*notMTpolicy C xs*) *else True*)
|*notMTpolicy C* [] = *False*

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

lemma *noneMT-vs-notMT*[*rule-format*]: *noneMT C p* ⟶ *p* ≠ [] ⟶ *notMTpolicy C p*
apply (*induct p*)
apply *simp-all*
done

lemma *RS3nMT*[*rule-format*]: *noneMT C* (*removeShadowRules3 C p*)
by (*induct p*,*simp-all*)

lemma *RS3nMT2*[*rule-format*]: *noneMT C p* ⟹ (*removeShadowRules3 C p*) = *p*

by (*induct p*,*simp-all*)

lemma *nMTcharn*: *noneMT C p* = (∀ *r* ∈ *set p*. *dom* (*C r*) ≠ {})
by (*induct p*) *simp-all*

lemma *nMTeqSet*: *set p* = *set s* ⟹ *noneMT C p* = *noneMT C s*
by (*simp add*: *nMTcharn*)

lemma *notMTnMT*: ⟦*a* ∈ *set p*; *noneMT C p*⟧ ⟹ *dom* (*C a*) ≠ {}
by (*simp add*: *nMTcharn*)

lemma *noneMTconc*[*rule-format*]: *noneMT C* (*a*@[*b*]) ⟶ *noneMT C a*
by (*induct a*, *simp-all*)

lemma *nMTtail*[*rule-format*]: *noneMT C p* ⟶ *noneMT C* (*tl p*)
by (*induct p*, *simp-all*)

lemma *notMTpolicyimpnotMT*[*simp*]: *notMTpolicy C p* ⟹ *p* ≠ []
by *auto*

lemma *SR3nMT*[*rule-format*]: ¬ *notMTpolicy C p* ⟶ *removeShadowRules3 C p* = []
by (*induct p*, *simp-all*)

lemma *NMPcharn*[*rule-format*]: *a* ∈ *set p*⟶ *dom* (*C a*) ≠ {} ⟶ *notMTpolicy C p*
by (*induct p*, *simp-all*)

lemma *NMPRS3*[*rule-format*]: *notMTpolicy C p* ⟶ *notMTpolicy C* (*removeShadowRules3 C p*)
by (*induct p*, *simp-all*)

Next, a few theorems about applied_rule

lemma *mrconc*[*rule-format*]: *applied-rule-rev C x p* = *Some a* ⟶
applied-rule-rev C x (*b*#*p*) = *Some a*
apply (*rule rev-induct*) **back**
apply (*simp-all add*: *applied-rule-rev-def*)
done

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*[*rule-format*]: *applied-rule-rev C x p* = *None* ⟶
applied-rule-rev C x (*b*#*p*) = *applied-rule-rev C x* [*b*]

apply (*rule-tac xs* = *p* **in** *rev-induct*)
apply *simp-all*
apply (*rule impI*)
apply (*case-tac x* ∈ *dom* (*C xa*))
apply (*simp-all add*: *applied-rule-rev-def*)
done

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*[*rule-format*]: *r* ∈ *set p* ⟶ *applied-rule-rev C x p* = *None* ⟶
x ∉ *dom* (*C r*)
apply (*rule rev-induct*, *simp-all*)
apply (*rule conjI*| *rule impI*)+
apply *simp-all*
apply (*case-tac xa* ∈ *set xs*)
apply (*simp-all add*: *applied-rule-rev-def split*: *if-splits*)
done

7.2 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* :: (′α ↦ ′β) ⇒ ((′γ ↦ ′δ) *list*) ⇒
((′α ↦ ′β) ⇒ (′γ ↦ ′δ) ⇒ ((′α × ′γ) ↦ (′β × ′δ))) ⇒
((′β × ′δ) ⇒ ′y) ⇒ (′x ⇒ (′α × ′γ)) ⇒
((′x ↦ ′y) *list*) (**infixr** ⨂$_L$ *54*) **where**
prod-list x (*y*#*ys*) *par-comb ran-adapt dom-adapt* =
((*ran-adapt o-f* ((*par-comb x y*) *o dom-adapt*))#(*prod-list x ys par-comb ran-adapt dom-adapt*))
| *prod-list x* [] *par-comb ran-adapt dom-adapt* = []

An instance, as usual there are four of them.

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

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

lemma *two-conc*: $(prod\text{-}list\ x\ (y\#ys)\ p\ r\ d) =$
$((r\ o\text{-}f\ ((p\ x\ y)\ o\ d))\#(prod\text{-}list\ x\ ys\ p\ r\ d))$
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*: $is\text{-}distr\ (op \bigotimes_{\vee D})\ d\ r$
apply (*simp add*: *is-distr-def*)
apply (*rule allI*)+
apply (*rule distr-orD*)
by *simp*

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

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

lemma *is-strict-2*: $is\text{-}strict\ (op \bigotimes_{2})\ d\ r$
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

declare *Fun.o-apply* [*simp del*]

The following theorems help to establish the main correctness theorem of the distribution. They are provided for all four parallel combination operators.

lemma *distr-aux1*[*rule-format*]:
$x \in dom\ (r\ o\text{-}f\ (A \circ d)) \longrightarrow$
$(r\ o\text{-}f\ (A \bigoplus B)\ o\ d)\ x = (r\ o\text{-}f\ (A \circ d))\ x$
by (*simp add*: *prod-2-def dom-def policy-range-comp-def map-add-def o-apply*
split: *option.splits decision.splits prod.splits*)

lemma *distr-aux2-1*[*rule-format*]: $x \in dom\ (r\ o\text{-}f\ ((P1 \bigotimes_2 a) \circ d)) \longrightarrow$
$((r\ o\text{-}f\ ((P1 \bigotimes_2 a \bigoplus Q) \circ d))\ x = (r\ o\text{-}f\ ((P1 \bigotimes_2 a) \circ d))\ x)$
apply (*simp add*: *prod-2-def dom-def policy-range-comp-def map-add-def o-apply*
split: *option.splits decision.splits prod.splits*)
done

lemma *distr-aux2-2*[*rule-format*]:
$x \notin dom\ (r\ o\text{-}f\ ((P1 \bigotimes_2 a) \circ d)) \longrightarrow$
$(((r\ o\text{-}f\ ((P1 \bigotimes_2 a \bigoplus b) \circ d))\ x) =$
$((r\ o\text{-}f\ ((P1 \bigotimes_2 b) \circ d))\ x))$
apply (*simp add*: *prod-2-def dom-def policy-range-comp-def map-add-def o-apply*
split: *option.splits decision.splits prod.splits*)
done

lemma *distr-aux1-1*[*rule-format*]: $x \in dom\ (r\ o\text{-}f\ ((P1 \bigotimes_1 a) \circ d)) \longrightarrow$
$((r\ o\text{-}f\ ((P1 \bigotimes_1 a \bigoplus Q) \circ d))\ x = (r\ o\text{-}f\ ((P1 \bigotimes_1 a) \circ d))\ x)$
apply (*simp add*: *prod-1-def dom-def policy-range-comp-def map-add-def o-apply*
split: *option.splits decision.splits prod.splits*)
done

lemma
$x \notin dom\ (\ (P1 \bigotimes_1 a) \circ d) \longrightarrow$
$(((((P1 \bigotimes_1 a \bigoplus b) \circ d))\ x) =$
$((((P1 \bigotimes_1 b) \circ d))\ x))$
apply (*simp add*: *prod-1-def dom-def policy-range-comp-def map-add-def o-apply*
split: *option.splits decision.splits prod.splits*)
done

lemma *distr-aux1-2*[*rule-format*]:
$x \notin dom\ (r\ o\text{-}f\ ((P1 \bigotimes_1 a) \circ d)) \longrightarrow$
$(((r\ o\text{-}f\ ((P1 \bigotimes_1 a \bigoplus b) \circ d))\ x) =$
$((r\ o\text{-}f\ ((P1 \bigotimes_1 b) \circ d))\ x))$
apply (*simp add*: *prod-1-def dom-def policy-range-comp-def map-add-def o-apply*
split: *option.splits decision.splits prod.splits*)
done

lemma *distr-auxA-1*[*rule-format*]: $x \in dom\ (r\ o\text{-}f\ ((P1 \bigotimes_{\vee A} a)\ o\ d)) \longrightarrow$
$((r\ o\text{-}f\ ((P1 \bigotimes_{\vee A} a \bigoplus Q) \circ d))\ x = (r\ o\text{-}f\ ((P1 \bigotimes_{\vee A} a) \circ d))\ x)$
apply (*simp add*: *prod-orA-def dom-def policy-range-comp-def map-add-def o-apply*
split: *option.splits decision.splits prod.splits*)
done

lemma
$x \notin dom\ (\ (P1 \bigotimes_{\vee A} a) \circ d) \longrightarrow$
$(((((P1 \bigotimes_{\vee A} a \bigoplus b) \circ d))\ x) =$
$((((P1 \bigotimes_{\vee A} b) \circ d))\ x))$
apply (*simp add*: *prod-orA-def dom-def policy-range-comp-def map-add-def o-apply*
split: *option.splits decision.splits prod.splits*)
done

lemma *distr-auxA-2*[*rule-format*]:
$x \notin dom\ (r\ o\text{-}f\ ((P1 \bigotimes_{\vee A} a) \circ d)) \longrightarrow$
$(((r\ o\text{-}f\ ((P1 \bigotimes_{\vee A} a \bigoplus b) \circ d))\ x) =$
$((r\ o\text{-}f\ ((P1 \bigotimes_{\vee A} b) \circ d))\ x))$
apply (*simp add*: *prod-orA-def dom-def policy-range-comp-def map-add-def o-apply*
split: *option.splits decision.splits prod.splits*)
done

lemma *distr-auxD-1*[*rule-format*]: $x \in dom\ (r\ o\text{-}f\ ((P1 \bigotimes_{\vee D} a) \circ d)) \longrightarrow$
$((r\ o\text{-}f\ ((P1 \bigotimes_{\vee D} a \bigoplus Q) \circ d))\ x = (r\ o\text{-}f\ ((P1 \bigotimes_{\vee D} a) \circ d))\ x)$
apply (*simp add*: *prod-orD-def dom-def policy-range-comp-def map-add-def o-apply*
split: *option.splits decision.splits prod.splits*)
done

lemma
$x \notin dom\ (\ (P1\ \bigotimes_{\vee D}\ a) \circ d) \longrightarrow$
$(((((P1\ \bigotimes_{\vee D}\ a\ \bigoplus\ b) \circ d))\ x) =$
$((((P1\ \bigotimes_{\vee D}\ b) \circ d))\ x))$
apply (*simp add*: *prod-orD-def dom-def policy-range-comp-def map-add-def o-apply*
split: *option.splits decision.splits prod.splits*)
done

lemma *distr-auxD-2*[*rule-format*]:
$x \notin dom\ (r\ o\text{-}f\ ((P1\ \bigotimes_{\vee D}\ a) \circ d)) \longrightarrow$
$(((r\ o\text{-}f\ ((P1\ \bigotimes_{\vee D}\ a\ \bigoplus\ b) \circ d))\ x) =$
$((r\ o\text{-}f\ \ ((P1\ \bigotimes_{\vee D}\ b) \circ d))\ x))$
apply (*simp add*: *prod-orD-def dom-def policy-range-comp-def map-add-def o-apply*
split: *option.splits decision.splits prod.splits*)
done

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

lemma *Norm-Distr-1*[*rule-format*]:
$((r\ o\text{-}f\ (((op\ \bigotimes_1)\ P1\ (list2policy\ P2))\ o\ d))\ x =$
$((list2policy\ ((P1\ \bigotimes_L\ P2)\ (op\ \bigotimes_1)\ r\ d))\ x))$
apply (*induct P2*)
apply (*simp add*: *policy-range-comp-def o-apply list2policy-def*)
apply (*simp add*: *list2policy-def*)
apply (*case-tac* $x \in dom\ (r\ o\text{-}f\ ((P1\ \bigotimes_1\ a) \circ d))$)
apply (*auto simp*: *domStart distr-aux1-1 distr-aux1-2 list2policy-def map-add-dom-app-simps*(*3*)
prod-2-list-def)
done

lemma *Norm-Distr-2*[*rule-format*]:
$((r\ o\text{-}f\ (((op\ \bigotimes_2)\ P1\ (list2policy\ P2))\ o\ d))\ x =$
$((list2policy\ ((P1\ \bigotimes_L\ P2)\ (op\ \bigotimes_2)\ r\ d))\ x))$
apply (*induct P2*)
apply (*simp add*: *policy-range-comp-def o-apply list2policy-def*)
apply (*simp add*: *list2policy-def*)
apply (*case-tac* $x \in dom\ (r\ o\text{-}f\ ((P1\ \bigotimes_2\ a) \circ d))$)
apply (*auto simp*: *domStart distr-aux2-1 distr-aux2-2 list2policy-def map-add-dom-app-simps*(*3*)
prod-2-list-def)
done

lemma *Norm-Distr-A*[*rule-format*]:
$((r\ o\text{-}f\ (((op\ \bigotimes_{\vee A})\ P1\ (list2policy\ P2))\ o\ d))\ x =$
$((list2policy\ ((P1\ \bigotimes_L\ P2)\ (op\ \bigotimes_{\vee A})\ r\ d))\ x))$
apply (*induct P2*)
apply (*simp add*: *policy-range-comp-def o-apply list2policy-def*)
apply (*simp add*: *list2policy-def*)
apply (*case-tac* $x \in dom\ (r\ o\text{-}f\ ((P1\ \bigotimes_{\vee A}\ a) \circ d))$)
apply (*auto simp*: *domStart distr-auxA-1 distr-auxA-2 list2policy-def map-add-dom-app-simps*(*3*)
prod-2-list-def)

done

lemma *Norm-Distr-D*[*rule-format*]:
((*r o-f* (((*op* $\bigotimes_{\vee D}$) *P1* (*list2policy P2*)) *o d*)) *x* =
((*list2policy* ((*P1* $\bigotimes_{L}$ *P2*) (*op* $\bigotimes_{\vee D}$) *r d*)) *x*))
apply (*induct P2*)
apply (*simp add*: *policy-range-comp-def o-apply list2policy-def*)
apply (*simp add*: *list2policy-def*)
apply (*case-tac x* ∈ *dom* (*r o-f* ((*P1* $\bigotimes_{\vee D}$ *a*) ∘ *d*)))
apply (*auto simp*: *domStart distr-auxD-1 distr-auxD-2 list2policy-def map-add-dom-app-simps*(*3*) *prod-2-list-def*)
done

Some domain reasoning

lemma *domSubsetDistr1*: *dom A* = *UNIV* ⟹ *dom* ((λ(*x*, *y*). *x*) *o-f* (*A* $\bigotimes_{1}$ *B*) *o* (λ *x*. (*x*,*x*)))
= *dom B*
apply (*rule set-eqI*)
apply (*rule iffI*)
by (*auto simp*: *prod-1-def policy-range-comp-def dom-def o-apply*
split: *decision.splits option.splits prod.splits*)

lemma *domSubsetDistr2*: *dom A* = *UNIV* ⟹ *dom* ((λ(*x*, *y*). *x*) *o-f* (*A* $\bigotimes_{2}$ *B*) *o* (λ *x*. (*x*,*x*)))
= *dom B*
apply (*rule set-eqI*)
apply (*rule iffI*)
by (*auto simp*: *prod-2-def policy-range-comp-def dom-def o-apply*
split: *decision.splits option.splits prod.splits*)

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

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

declare *o-apply* [*simp*]

end

theory *NormalisationTestSpecification*
imports *Normalisation*
begin

7.3 Using Transformations for Testing

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 establies 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*[*rule-format*]:
$x \in dom\ P \longrightarrow (list2policy\ PL)\ x = P\ x\ \longrightarrow$
$(PUTList\ PUT\ x\ PL \longrightarrow (PUT\ x = P\ x))$
apply (*induct-tac PL*)
by (*auto simp*: *list2policy-def dom-def*)

lemma *PUTList-None*[*rule-format*]:
$x \notin dom\ (list2policy\ list) \longrightarrow PUTList\ PUT\ x\ list$
apply (*induct-tac list*)
by (*auto simp*: *list2policy-def dom-def*)

lemma *PUTList-DomMT*[*rule-format*]:
$(\forall\, y{\in}set\ list.\ dom\ a \cap dom\ y = \{\}) \longrightarrow x \in (dom\ a) \longrightarrow$
$x \notin dom\ (list2policy\ list)$
apply (*induct-tac list*)

by (*auto simp*: *dom-def list2policy-def*)

lemma *distrPUTL2*[*rule-format*]:
$x \in dom\ P \longrightarrow (list2policy\ PL)\ x = P\ x \longrightarrow disjDom\ PL \longrightarrow$
$(PUT\ x = P\ x) \longrightarrow PUTList\ PUT\ x\ PL$
apply (*induct-tac PL*)
apply (*simp-all add*: *list2policy-def*)
apply *auto*
apply (*case-tac* $x \in dom\ a$)
apply (*case-tac list2policy list* $x = P\ x$, *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

lemma *distrPUTL*[*rule-format*]:
$[\![x \in dom\ P;\ (list2policy\ PL)\ x = P\ x;\ disjDom\ PL]\!] \Longrightarrow$
$(PUT\ x = P\ x) = PUTList\ PUT\ x\ PL$
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**
$gatherDomain\text{-}aux\ (x\#xs) = (dom\ x \cup (gatherDomain\text{-}aux\ xs))$
$|gatherDomain\text{-}aux\ [] = \{\}$

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*[*rule-format*]:
$x \in dom\ P \longrightarrow (list2policy\ PL)\ x = P\ x \longrightarrow$
$(PUTListGD\ PUT\ x\ PL \longrightarrow (PUT\ x = P\ x))$
apply (*simp add*: *PUTListGD-def disjDomGD-def gatherDomain-def list2policy-def*)
apply (*induct-tac PL*)
apply (*metis foldr.simps*(*1*) *notDom*)
by (*auto simp*: *dom-def*)

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

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

end