# HOL-TestGenFW

Achim D. Brucker Lukas Brügger Burkhart Wolff

October 15, 2012

## Contents

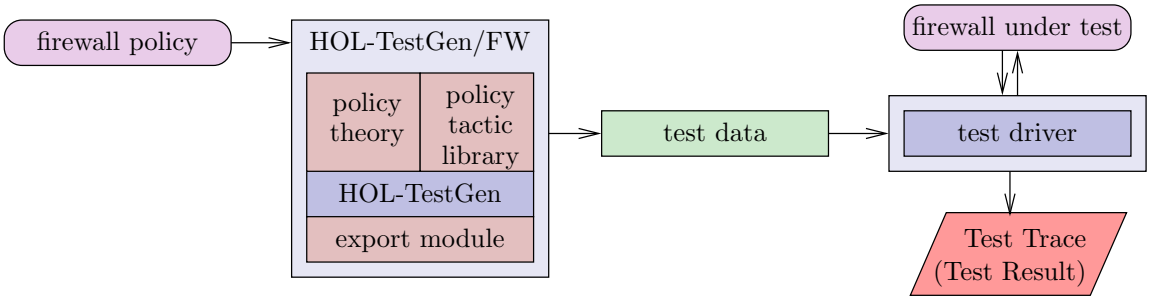


Figure 1: The HOL-TestGen/FW architecture.



# 1. Introduction

As HOL-TestGen is built on the framework of Isabelle with a general plug-in mechanism, HOL-TestGen can be customized to implement domain-specific, model-based test tools in its own right. As an example for such a domain-specific test-tool, we developed HOL-TestGen/FW which extends HOL-TestGen by:

1. a theory (or library) formalising networks, protocols and firewall policies,
2. domain-specific extensions of the generic test-case procedures (tactics), and
3. support for an export format of test-data for external tools such as [4].

HOL-TestGen/FW is part of the HOL-TestGen distribution. It is located in the directory `add-ons/security`; see [3, 2] for more details.

Figure 1 shows the overall architecture of HOL-TestGen/FW.

In fact, item 1 defines the formal semantics (in HOL) of a specification language for firewall policies; see [2] and the accompanying examples for details. On the technical level, this library also contains simplification rules together with the corresponding setup of the constraint resolution procedures.

With item 2 we refer to domain-specific processing encapsulated into the general HOL-TestGen test-case generation. Since test specifications in our domain have a specific pattern consisting of a limited set of predicates and policy combinators, this can be exploited in specific pre-processing and post-processing of an optimised version of the procedure, now tuned for stateless firewall policies.

With item 3, we refer to an own XML-like format for exchanging test-data for firewalls, i.e. a description of packets to be send together with the expected behavior of the firewall. This data data can be imported in a test-driver for firewalls, for example [4]. This completes our toolchain which, thus, supports the execution of test data on firewall implementations based on test cases derived from formal specifications.

## 2. Installing and using HOL-TestGen/FW

To install HOL-TestGen/FW you need a working installation of HOL-TestGen as described in the HOL-TestGen User Guide. To build the extension, go into the `add-ons/security/src/firew` directory and build the HOL-TestGen/FW heap image for Isabelle by calling

```
isabelle make
```

Note that this requires that the HOL-UPF heap has been built before. HOL-TestGen/FW can now be started using the `isabelle` command:

```
isabelle jedit -l HOL-TestGenFW
```

or, if HOL-TestGen was built on top of HOLCF instead on HOL only:

```
isabelle jedit -l HOLCF-TestGenFW
```

## 3. Preliminaries

**theory**
*FWTesting*
**imports**
*PacketFilter/PacketFilter*
*NAT/NAT*
*FWNormalisation/FWNormalisation*
*StatefulFW/StatefulFW*
**begin**

**declare** *dom-eq-empty-conv* [*simp del*]

This is the formalisation in Isabelle/HOL of firewall policies and corresponding networks and packets. It first contains the formalisation of stateless packet filters as described in [2], followed by a verified policy normalisation technique (described in [1]), and a formalisation of stateful protocols described in [3].

The following statement adjusts the pre-normalization step of the default test case generation algorithm. This turns out to be more efficient for the specific case of firewall policies.

Next, the Isar command *prepare-fw-spec* is specified. It can be used to turn test specifications of the form: "$C\ x \Longrightarrow FUT\ x\ =\ policy\ x$" into the desired form for test case generation.

⟨*ML*⟩

This theory contains a collection of (rather unsorted lemmas) which are of general use when processing test specification (including Normalization) of the Firewall Testing heap.

**definition** *policyID* **where**
$policyID = (\lambda\ x.\ (Some\ (allow\ (x))))$

**lemma** *setDistinct*: $[\![x \in a;\ x \notin b]\!] \Longrightarrow a \neq b$
⟨*proof*⟩

**lemma** *setDistinct3*: $[\![x \neq y]\!] \Longrightarrow$
$\{\{(a,\ b,\ c).\ a \in x\}\} \neq \{\{(a,\ b,\ c).\ a \in y\}\}$
⟨*proof*⟩

**lemma** *setDistinct4*: $[\![(a::int) \neq x;\ a < b;\ x < y]\!]$
$\Longrightarrow \{a..b\} \neq \{x..y\}$
⟨*proof*⟩

**lemma** *setDistinct2*: $[\![x \neq y]\!] \Longrightarrow$
$\{(a,\ b).\ a \in x\} \neq \{(a,\ b).\ a \in y\}$
⟨*proof*⟩

**lemma** *setDistinct1*: $[\![x \neq y]\!] \Longrightarrow$
$\{(a,\ b,\ c).\ a \in x\} \neq \{(a,\ b,\ c).\ a \in y\}$
⟨*proof*⟩

**lemma** *AllowNMT*: $[\![x \neq \{\};\ y \neq \{\}]\!] \Longrightarrow$
$dom\ (Cp\ (AllowPortFromTo\ \{\{(a,\ b,\ c).\ a \in x\}\}\ \{\{(a,\ b,\ c).\ a \in y\}\}\ (p,i))) \neq \{\}$
⟨*proof*⟩

**lemma** *domDA1Cp*: $dom\ (Cp\ (DenyAll \oplus P)) = UNIV$
⟨*proof*⟩

**lemma** *aux*: $[\![x \neq a;\ y \neq b;\ (x \neq y \wedge x \neq b) \vee\ (a \neq b \wedge a \neq y)]\!] \Longrightarrow \{x,a\} \neq \{y,b\}$
⟨*proof*⟩

**lemma** *aux2*: $\{a,b\} = \{b,a\}$
⟨*proof*⟩

**lemma** *deny-comm*[*simp*]: $(deny() = X) = (X = deny())$
⟨*proof*⟩

**lemma** *allow-comm*[*simp*]: $(allow() = X) = (X = allow())$
⟨*proof*⟩

**lemma** *PO-add*: $PO\ P \Longrightarrow P$
⟨*proof*⟩

**lemma** *dom-2-subset1*: $dom\ (r\ o\text{-}f\ ((A \bigotimes_2 B)\ o\ (\lambda\ x.\ (x,x)))) \subseteq dom\ A$
⟨*proof*⟩

**lemma** *dom-2-comm*: $dom\ (A \bigotimes_2 B) = dom\ (B \bigotimes_2 A)$
⟨*proof*⟩

**lemma** *dom-2-subset2*: $dom\ (r\ o\text{-}f\ ((A \bigotimes_2 B)\ o\ (\lambda\ x.\ (x,x)))) \subseteq dom\ B$
⟨*proof*⟩

**lemma** *dom-2*: $dom\ A = UNIV \Longrightarrow$
$dom\ (r\ o\text{-}f\ ((A \bigotimes_2 B)\ o\ (\lambda\ x.\ (x,x)))) = dom\ B$
⟨*proof*⟩

**lemma** *domDA1*: $dom\ (C\ (DenyAll \oplus P)) = UNIV$
⟨*proof*⟩

**lemma** *if-DA-simp*: $((if\ P\ then\ (deny\text{-}all\ x)\ else\ None) =$
$None) = (\neg\ P)$
⟨*proof*⟩

**lemma** *if-DA-simp2*: $(\exists\, y.\ (if\ P\ then\ deny\text{-}all\ x\ else\ None) = Some\ y) = P$
⟨*proof*⟩

**lemma** *if-DA-simp3*: $((if\ P\ then\ deny\text{-}all\ x\ else\ None) =$
$Some\ ae) = (P \wedge deny\text{-}all\ x = Some\ ae)$
⟨*proof*⟩

**lemma** *if-DA-simp4*: $((if\ P$

*then Some (allow x)*
*else None) =*
*None) = (¬ P)*
⟨*proof*⟩

**lemma** *if-DA-simp5*: ((*if P*
*then Some (deny x)*
*else None) =*
*None) = (¬ P)*
⟨*proof*⟩

**lemma** *setDistinct6*: ⟦$x \neq y$⟧ ⟹
$\{(a, b, c).\ a = x\} \neq \{(a, b, c).\ a = y\}$
⟨*proof*⟩

**definition** *ip2int* :: (*int* × *int* × *int* × *int*) ⇒ *int* **where**
*ip2int x* = (((*fst x*)∗*16777216*) + ((*fst* (*snd x*))∗*65536*) +
((*fst* (*snd* (*snd x*)))∗*256*) + *snd* (*snd* (*snd x*)))

**lemmas** *if-DA* = *if-DA-simp if-DA-simp2 if-DA-simp3 if-DA-simp4 if-DA-simp5*

**declare** *if-DA* [*simp*]

**end**

# 4. Packets and Networks

**theory** *NetworkCore*
**imports** *Main UPF*
**begin**

In networks based e.g. on TCP/IP, a message from A to B is encapsulated in *packets*, which contain the content of the message and routing information. The routing information mainly contains its source and its destination address.

In the case of stateless packet filters, a firewall bases its decision upon this routing information and, in the stateful case, on the content. Thus, we model a packet as a four-tuple of the mentioned elements, together with an id field.

The ID is an integer:

**type-synonym** *id* = *int*

To enable different representations of addresses (e.g. IPv4 and IPv6, with or without

ports), we model them as an unconstrained type class and directly provide several instances:

**class** *adr*

**type-synonym** $'\alpha\ src = '\alpha$
**type-synonym** $'\alpha\ dest = '\alpha$

**instance** *int* ::*adr* ⟨*proof*⟩
**instance** *nat* ::*adr* ⟨*proof*⟩

**instance** *fun* :: (*adr*,*adr*) *adr* ⟨*proof*⟩
**instance** *prod* :: (*adr*,*adr*) *adr* ⟨*proof*⟩

The content is also specified with an unconstrained generic type:

**type-synonym** $'\beta\ content = '\beta$

For applications where the concrete representation of the content field does not matter (usually the case for stateless packet filters), we provide a default type which can be used in those cases:

**datatype** *DummyContent* = *data*

Finally, a packet is:

**type-synonym** $('\alpha,'\beta)\ packet = id \times '\alpha\ src \times '\alpha\ dest \times '\beta\ content$

Please note that protocols (e.g. http) are not modelled explicitly. In the case of stateless packet filters, they are only visible by the destination port of a packet, which are modelled as part of the address. Additionally, stateful firewalls often determine the protocol by the content of a packet.

**definition** $src :: ('\alpha::adr,'\beta)\ packet \Rightarrow '\alpha$
**where** $src = fst\ o\ snd$

Port numbers (which are part of an address) are also modelled in a generic way. The integers and the naturals are typical representations of port numbers.

**class** *port*

**instance** *int* ::*port* ⟨*proof*⟩
**instance** *nat* :: *port* ⟨*proof*⟩
**instance** *fun* :: (*port*,*port*) *port* ⟨*proof*⟩
**instance** *prod* :: (*port*,*port*) *port* ⟨*proof*⟩

A packet therefore has two parameters, the first being the address, the second the content. For the sake of simplicity, we do not allow to have a different address representation format for the source and the destination of a packet.

In order to access the different parts of a packet directly, we define a couple of projectors:

**definition** *id* :: (′α::*adr*,′β) *packet* ⇒ *id*
**where** *id* = *fst*

**definition** *dest* :: (′α::*adr*,′β) *packet* ⇒ ′α *dest*
**where** *dest* = *fst o snd o snd*

**definition** *content* :: (′α::*adr*,′β) *packet* ⇒ ′β *content*
**where** *content* = *snd o snd o snd*

**datatype** *protocol* = *tcp* | *udp*

**lemma** *either*: ⟦*a* ≠ *tcp*;*a* ≠ *udp*⟧ ⟹ *False*
⟨*proof*⟩

**lemma** *either2*[*simp*]: (*a* ≠ *tcp*) = (*a* = *udp*)
⟨*proof*⟩

**lemma** *either3*[*simp*]: (*a* ≠ *udp*) = (*a* = *tcp*)
⟨*proof*⟩

The following two constants give the source and destination port number of a packet. Address representations using port numbers need to provide a definition for these types.

**consts** *src-port* :: (′α::*adr*,′β) *packet* ⇒ ′γ::*port*
**consts** *dest-port* :: (′α::*adr*,′β) *packet* ⇒ ′γ::*port*
**consts** *src-protocol* :: (′α::*adr*,′β) *packet* ⇒ *protocol*
**consts** *dest-protocol* :: (′α::*adr*,′β) *packet* ⇒ *protocol*

A subnetwork (or simply a network) is a set of sets of addresses.

**type-synonym** ′α *net* = ′α *set set*

The relation in_subnet (⊏) checks if an address is in a specific network.

**definition**
*in-subnet* :: ′α::*adr* ⇒ ′α *net* ⇒ *bool* (**infixl** ⊏ *100*) **where**
*in-subnet a S* = (∃ *s* ∈ *S*. *a* ∈ *s*)

The following lemmas will be useful later.

**lemma** *in-subnet*:
(*a*, *e*) ⊏ {{(*x1*,*y*). *P x1 y*}} = *P a e*
⟨*proof*⟩

**lemma** *src-in-subnet*:
*src*(*q*,(*a*,*e*),*r*,*t*) ⊏ {{(*x1*,*y*). *P x1 y*}} = *P a e*
⟨*proof*⟩

**lemma** *dest-in-subnet*:
*dest* (*q*,*r*,((*a*),*e*),*t*) ⊏ {{(*x1*,*y*). *P x1 y*}} = *P a e*
⟨*proof*⟩

Address models should provide a definition for the following constant, returning a network consisting of the input address only.

**consts** *subnet-of* :: 'α::*adr* ⇒ 'α *net*

**lemmas** *packet-defs* = *in-subnet-def id-def content-def src-def dest-def*

**end**

# 5. Address Representations

**theory**
*NetworkModels*
**imports**

*DatatypeAddress*
*DatatypePort*

*IntegerAddress*
*IntegerPort*
*IntegerPort-TCPUDP*

*IPv4*
*IPv4-TCPUDP*
**begin**

One can think of many different possible address representations. In this distribution, we include seven different variants:

- DatatypeAddress: Three explicitly named addresses, which build up a network consisting of three disjunct subnetworks. I.e. there are no overlaps and there is no way to distinguish between individual hosts within a network.
- DatatypePort: An address is a pair, with the first element being the same as above, and the second being a port number modelled as an Integer[1].
- adr_i: An address in an Integer.
- adr_ip: An address is a pair of an Integer and a port (which is again an Integer).

[1]For technical reasons, we always use Integers instead of Naturals. As a consequence, the test specifications have to be adjusted to eliminate negative numbers.

- adr_ipp: An address is a triple consisisting of two Integers modelling the IP address and the port number, and the specification of the network protocol
- IPv4: An address is a pair. The first element is a four-tuple of Integers, modelling an IPv4 address, the second element is an Integer denoting the port number.
- IPv4_TCPUDP: The same as above, but including additionally the speficiation of the network protocol.

The theories of each pf the networks are relatively small. It suffices to provide the required types, a couple of lemmas, and - if required - a definition for the source and destination ports of a packet.

**end**

## 5.1. Datatype Addresses

**theory** *DatatypeAddress*
**imports** *NetworkCore*
**begin**

A theory describing a network consisting of three subnetworks. Hosts within a network are not distinguished.

**datatype** *DatatypeAddress* = *dmz-adr* | *intranet-adr* | *internet-adr*

**definition**
*dmz*::*DatatypeAddress net* **where**
*dmz* = {{*dmz-adr*}}
**definition**
*intranet*::*DatatypeAddress net* **where**
*intranet* = {{*intranet-adr*}}
**definition**
*internet*::*DatatypeAddress net* **where**
*internet* = {{*internet-adr*}}

**end**

## 5.2. Datatype Addresses with Ports

**theory** *DatatypePort*
**imports** *NetworkCore*
**begin**

A theory describing a network consisting of three subnetworks, including port numbers modelled as Integers. Hosts within a network are not distinguished.

**datatype** *DatatypeAddress* = *dmz-adr* | *intranet-adr* | *internet-adr*

**type-synonym**
*port* = *int*
**type-synonym**
*DatatypePort* = (*DatatypeAddress* × *port*)

**instance** *DatatypeAddress* :: *adr* ⟨*proof*⟩

**definition**
*dmz*::*DatatypePort net* **where**
*dmz* = {{(*a*,*b*). *a* = *dmz-adr*}}
**definition**
*intranet*::*DatatypePort net* **where**
*intranet* = {{(*a*,*b*). *a* = *intranet-adr*}}
**definition**
*internet*::*DatatypePort net* **where**
*internet* = {{(*a*,*b*). *a* = *internet-adr*}}

**defs** (**overloaded**)
*src-port-def*: *src-port* (*x*::(*DatatypePort*,′*β*) *packet*) ≡ (*snd o fst o snd*) *x*
*dest-port-def*: *dest-port* (*x*::(*DatatypePort*,′*β*) *packet*) ≡(*snd o fst o snd o snd*) *x*
*subnet-of-def*: *subnet-of* (*x*::*DatatypePort*) ≡ {{(*a*,*b*). *a* = *fst x*}}

**lemma** *src-port* : *src-port* ((*a*,*x*,*d*,*e*)::(*DatatypePort*,′*β*) *packet*) = *snd x*
⟨*proof*⟩

**lemma** *dest-port* : *dest-port* ((*a*,*d*,*x*,*e*)::(*DatatypePort*,′*β*) *packet*) = *snd x*
⟨*proof*⟩

**lemmas** *DatatypePortLemmas* = *src-port dest-port src-port-def dest-port-def*

**end**

## 5.3. Integer Addresses

**theory** *IntegerAddress*
**imports** *NetworkCore*
**begin**

A theory where addresses are modelled as Integers.

**type-synonym**

$adr_i = int$

**end**

## 5.4. Integer Addresses with Ports

**theory** *IntegerPort*
**imports** *NetworkCore*
**begin**

A theory describing addresses which are modelled as a pair of Integers - the first being the host address, the second the port number.

**type-synonym**
$address = int$

**type-synonym**
$port = int$

**type-synonym**
$adr_{ip} = address \times port$

**defs** (**overloaded**)
*src-port-def*: $src\text{-}port\ (x{::}(adr_{ip},'\beta)\ packet) \equiv (snd\ o\ fst\ \ o\ snd)\ x$
*dest-port-def*: $dest\text{-}port\ \ (x{::}(adr_{ip},'\beta)\ packet) \equiv (snd\ o\ fst\ o\ \ snd\ o\ snd)\ x$
*subnet-of-def*: $subnet\text{-}of\ \ (x{::}(adr_{ip})) \equiv \{\{(a,b).\ a = fst\ x\}\}$

**lemma** *src-port*: $src\text{-}port\ (a,x{::}adr_{ip},d,e) = snd\ x$
⟨*proof*⟩

**lemma** *dest-port*: $dest\text{-}port\ (a,d,x{::}adr_{ip},e) = snd\ x$
⟨*proof*⟩

**lemmas** $adr_{ip}Lemmas = src\text{-}port\ dest\text{-}port\ src\text{-}port\text{-}def\ dest\text{-}port\text{-}def$

**end**

## 5.5. Integer Addresses with Ports and Protocols

**theory** *IntegerPort-TCPUDP*
**imports** *NetworkCore*

**begin**

A theory describing addresses which are modelled as a pair of Integers - the first being the host address, the second the port number.

**type-synonym**
*address* = *int*

**type-synonym**
*port* = *int*

**type-synonym**
$adr_{ipp}$ = *address* × *port* × *protocol*

**instance** *protocol* :: *adr* ⟨*proof*⟩

**defs** (**overloaded**)
*src-port-def*: *src-port* (*x*::($adr_{ipp}$,'β) *packet*) ≡ (*fst o snd o fst o snd*) *x*
*dest-port-def*: *dest-port* (*x*::($adr_{ipp}$,'β) *packet*) ≡ (*fst o snd o fst o snd o snd*) *x*
*subnet-of-def*: *subnet-of* (*x*::($adr_{ipp}$)) ≡ {{(*a*,*b*,*c*). *a* = *fst x*}}
*src-protocol-def*: *src-protocol* (*x*::($adr_{ipp}$,'β) *packet*) ≡ (*snd o snd o fst o snd*) *x*
*dest-protocol-def*: *dest-protocol* (*x*::($adr_{ipp}$,'β) *packet*) ≡ (*snd o snd o fst o snd o snd*) *x*

**lemma** *src-port*: *src-port* (*a*,*x*::$adr_{ipp}$,*d*,*e*) = *fst* (*snd x*)
⟨*proof*⟩

**lemma** *dest-port*: *dest-port* (*a*,*d*,*x*::$adr_{ipp}$,*e*) = *fst* (*snd x*)
⟨*proof*⟩

Common test constraints:

**definition** *port-positive* :: ($adr_{ipp}$,'*b*) *packet* ⇒ *bool* **where**
*port-positive x* = (*dest-port x* > (*0*::*port*))

**definition** *fix-values* :: ($adr_{ipp}$,*DummyContent*) *packet* ⇒ *bool* **where**
*fix-values x* = (*src-port x* = (*1*::*port*) ∧ *src-protocol x* = *udp* ∧ *content x* = *data* ∧ *id x* = *1*)

**lemmas** $adr_{ipp}$*Lemmas* = *src-port dest-port src-port-def dest-port-def src-protocol-def dest-protocol-def subnet-of-def*

**lemmas** $adr_{ipp}$*TestConstraints* = *port-positive-def fix-values-def*

**end**

### 5.6. IPv4 Addresses

**theory** *IPv4*
**imports** *NetworkCore*
**begin**

A theory describing IPv4 addresses with ports. The host address is a four-tuple of Integers, the port number is a single Integer.

**type-synonym**
*ipv4-ip* = (*int* × *int* × *int* × *int*)

**type-synonym**
*port* = *int*

**type-synonym**
*ipv4* = (*ipv4-ip* × *port*)

**defs** (**overloaded**)
*src-port-def*: *src-port* (*x*::(*ipv4*,'*β*) *packet*) ≡ (*snd o fst o snd*) *x*
**defs** (**overloaded**)
*dest-port-def*:*dest-port* (*x*::(*ipv4*,'*β*) *packet*) ≡ (*snd o fst o snd o snd*) *x*
**defs** (**overloaded**)
*subnet-of-def*: *subnet-of* (*x*::*ipv4*) ≡ {{(*a*,*b*). *a* = *fst x*}}

**definition** *subnet-of-ip* :: *ipv4-ip* ⇒ *ipv4 net*
**where** *subnet-of-ip ip* = {{(*a*,*b*). (*a* = *ip*)}}

**lemma** *src-port*: *src-port* (*a*,(*x*::*ipv4*),*d*,*e*) = *snd x*
⟨*proof*⟩

**lemma** *dest-port*: *dest-port* (*a*,*d*,(*x*::*ipv4*),*e*) = *snd x*
⟨*proof*⟩

**lemmas** *IPv4Lemmas* = *src-port dest-port src-port-def dest-port-def*

**end**

# 6. Policies

## 6.1. Policy Core

**theory** *PolicyCore*
**imports** *NetworkCore UPF*
**begin**

A policy is seen as a partial mapping from packet to packet out.

**type-synonym** $('\alpha, '\beta)$ *FWPolicy* = $('\alpha, '\beta)$ *packet* $\mapsto$ *unit*

When combining several rules, the firewall is supposed to apply the first matching one. In our setting this means the first rule which maps the packet in question to *Some* (*packet out*). This is exactly what happens when using the map-add operator (*rule1* ++ *rule2*). The only difference is that the rules must be given in reverse order.

The constant *p-accept* is *True* iff the policy accepts the packet.

**definition**
*p-accept* :: $('\alpha, '\beta)$ *packet* $\Rightarrow$ $('\alpha, '\beta)$ *FWPolicy* $\Rightarrow$ *bool* **where**
*p-accept p pol* = (*pol p* = $\lfloor$*allow* ()$\rfloor$)

**end**

## 6.2. Policy Combinators

**theory** *PolicyCombinators*
**imports**
*PolicyCore*
**begin**

In order to ease the specification of a concrete policy, we define some combinators. Using these combinators, the specification of a policy gets very easy, and can be done similarly as in tools like IPTables.

**definition**
*allow-all-from* :: $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*allow-all-from src-net* = {*pa*. *src pa* $\sqsubset$ *src-net*} $\triangleleft$ $A_U$

**definition**
*deny-all-from* :: $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*deny-all-from src-net* = {*pa*. *src pa* $\sqsubset$ *src-net*} $\triangleleft D_U$

**definition**
*allow-all-to* :: $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*allow-all-to dest-net* = {*pa*. *dest pa* $\sqsubset$ *dest-net*} $\triangleleft$ $A_U$

**definition**

*deny-all-to* :: $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*deny-all-to dest-net* = {*pa*. *dest pa* $\sqsubset$ *dest-net*} $\triangleleft D_U$

**definition**
*allow-all-from-to* :: $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*allow-all-from-to src-net dest-net* =
{*pa*. *src pa* $\sqsubset$ *src-net* $\wedge$ *dest pa* $\sqsubset$ *dest-net*} $\triangleleft A_U$

**definition**
*deny-all-from-to* :: $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*deny-all-from-to src-net dest-net* =
{*pa*. *src pa* $\sqsubset$ *src-net* $\wedge$ *dest pa* $\sqsubset$ *dest-net*} $\triangleleft D_U$

All these combinators and the default rules are put into one single lemma called *Policy-Combinators* to faciliate proving over policies.

**lemmas** *PolicyCombinators* = *allow-all-from-def deny-all-from-def*
*allow-all-to-def deny-all-to-def allow-all-from-to-def*
*deny-all-from-to-def UPFDefs*

**end**

## 6.3. Policy Combinators with Ports

**theory** *PortCombinators*
**imports** *PolicyCombinators*
**begin**

This theory defines policy combinators for those network models which have ports. They are provided in addition to the the ones defined in the PolicyCombinators theory.

This theory requires from the network models a definition for the two following constants:

- $src\_port :: (\alpha, \beta) packet \Rightarrow (\gamma :: port)$
- $dest\_port :: (\alpha, \beta) packet \Rightarrow (\gamma :: port)$

**definition**
*allow-all-from-port* :: $'\alpha$::*adr net* $\Rightarrow$ ($'\gamma$::*port*) $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*allow-all-from-port src-net s-port* =
{*pa*. *src-port pa* = *s-port*} $\triangleleft$ *allow-all-from src-net*

**definition**
*deny-all-from-port* :: $'\alpha$::*adr net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*deny-all-from-port src-net s-port* =
{*pa*. *src-port pa* = *s-port*} $\triangleleft$ *deny-all-from src-net*

**definition**
*allow-all-to-port* :: $'\alpha$::*adr net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*allow-all-to-port dest-net d-port* =
{*pa*. *dest-port pa* = *d-port*} $\triangleleft$ *allow-all-to dest-net*

**definition**
*deny-all-to-port* :: $'\alpha$::*adr net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*deny-all-to-port dest-net d-port* =
{*pa*. *dest-port pa* = *d-port*} $\triangleleft$ *deny-all-to dest-net*

**definition**
*allow-all-from-port-to*:: $'\alpha$::*adr net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*)
**where**
*allow-all-from-port-to src-net s-port dest-net*
= {*pa*. *src-port pa* = *s-port*} $\triangleleft$ *allow-all-from-to src-net dest-net*

**definition**
*deny-all-from-port-to*::$'\alpha$::*adr net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*)
**where**
*deny-all-from-port-to src-net s-port dest-net*
= {*pa*. *src-port pa* = *s-port*} $\triangleleft$ *deny-all-from-to src-net dest-net*

**definition**
*allow-all-from-port-to-port*::$'\alpha$::*adr net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$
(($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*allow-all-from-port-to-port src-net s-port dest-net d-port* =
{*pa*. *dest-port pa* = *d-port*} $\triangleleft$ *allow-all-from-port-to src-net s-port dest-net*

**definition**
*deny-all-from-port-to-port* :: $'\alpha$::*adr net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$
$'\gamma$::*port* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*deny-all-from-port-to-port src-net s-port dest-net d-port* =
{*pa*. *dest-port pa* = *d-port*} $\triangleleft$ *deny-all-from-port-to src-net s-port dest-net*

**definition**
*allow-all-from-to-port* :: $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$
$'\gamma$::*port* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*allow-all-from-to-port src-net dest-net d-port* =
{*pa*. *dest-port pa* = *d-port*} $\triangleleft$ *allow-all-from-to src-net dest-net*

**definition**
*deny-all-from-to-port* :: $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$
(($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*deny-all-from-to-port src-net dest-net d-port* = {*pa*. *dest-port pa* = *d-port*} $\triangleleft$ *deny-all-from-to src-net dest-net*

**definition**
*allow-from-port-to* :: $'\gamma$::*port* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*)
**where**

*allow-from-port-to port src-net dest-net* =
$\{pa.\ src\text{-}port\ pa = port\} \triangleleft allow\text{-}all\text{-}from\text{-}to\ src\text{-}net\ dest\text{-}net$

**definition**
*deny-from-port-to* :: $'\gamma$::*port* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*)
**where**
*deny-from-port-to port src-net dest-net* =
$\{pa.\ src\text{-}port\ pa = port\} \triangleleft deny\text{-}all\text{-}from\text{-}to\ src\text{-}net\ dest\text{-}net$

**definition**
*allow-from-to-port* :: $'\gamma$::*port* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*)
**where**
*allow-from-to-port port src-net dest-net* =
$\{pa.\ dest\text{-}port\ pa = port\} \triangleleft allow\text{-}all\text{-}from\text{-}to\ src\text{-}net\ dest\text{-}net$

**definition**
*deny-from-to-port* :: $'\gamma$::*port* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*)
**where**
*deny-from-to-port port src-net dest-net* =
$\{pa.\ dest\text{-}port\ pa = port\} \triangleleft deny\text{-}all\text{-}from\text{-}to\ src\text{-}net\ dest\text{-}net$

**definition**
*allow-from-ports-to* :: $'\gamma$::*port set* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$
(($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*allow-from-ports-to ports src-net dest-net* =
$\{pa.\ src\text{-}port\ pa \in ports\} \triangleleft allow\text{-}all\text{-}from\text{-}to\ src\text{-}net\ dest\text{-}net$

**definition**
*allow-from-to-ports* :: $'\gamma$::*port set* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$
(($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*allow-from-to-ports ports src-net dest-net* =
$\{pa.\ dest\text{-}port\ pa \in ports\} \triangleleft allow\text{-}all\text{-}from\text{-}to\ src\text{-}net\ dest\text{-}net$

**definition**
*deny-from-ports-to* :: $'\gamma$::*port set* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$
(($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*deny-from-ports-to ports src-net dest-net* =
$\{pa.\ src\text{-}port\ pa \in ports\} \triangleleft deny\text{-}all\text{-}from\text{-}to\ src\text{-}net\ dest\text{-}net$

**definition**
*deny-from-to-ports* :: $'\gamma$::*port set* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$
(($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
*deny-from-to-ports ports src-net dest-net* =
$\{pa.\ dest\text{-}port\ pa \in ports\} \triangleleft deny\text{-}all\text{-}from\text{-}to\ src\text{-}net\ dest\text{-}net$

**definition**
*allow-all-from-port-tos*:: $'\alpha$::*adr net* $\Rightarrow$ ($'\gamma$::*port*) *set* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*)
**where**
*allow-all-from-port-tos src-net s-port dest-net*
= {*pa*. *dest-port pa* $\in$ *s-port*} $\triangleleft$ *allow-all-from-to src-net dest-net*

As before, we put all the rules into one lemma called PortCombinators to ease writing later.

**lemmas** *PortCombinatorsCore* =

*allow-all-from-port-def deny-all-from-port-def allow-all-to-port-def*
*deny-all-to-port-def allow-all-from-to-port-def*
*deny-all-from-to-port-def*
*allow-from-ports-to-def allow-from-to-ports-def*
*deny-from-ports-to-def deny-from-to-ports-def*
*allow-all-from-port-to-def deny-all-from-port-to-def*
*allow-from-port-to-def allow-from-to-port-def deny-from-to-port-def*
*deny-from-port-to-def allow-all-from-port-tos-def*

**lemmas** *PortCombinators* =
*PortCombinatorsCore PolicyCombinators*

**end**

## 6.4. Policy Combinators with Ports and Protocols

**theory** *ProtocolPortCombinators*
**imports** *PortCombinators*
**begin**

This theory defines policy combinators for those network models which have ports. They are provided in addition to the the ones defined in the PolicyCombinators theory.

This theory requires from the network models a definition for the two following constants:

- $src\_port :: (\alpha, \beta)packet \Rightarrow (\gamma :: port)$
- $dest\_port :: (\alpha, \beta)packet \Rightarrow (\gamma :: port)$

**definition**
*allow-all-from-port-prot* :: *protocol* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ ($'\gamma$::*port*) $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*)
**where**
*allow-all-from-port-prot p src-net s-port* =
{*pa*. *dest-protocol pa* = *p*} $\triangleleft$ *allow-all-from-port src-net s-port*

**definition**
*deny-all-from-port-prot* :: *protocol* => 'α::*adr net* ⇒ 'γ::*port* ⇒ (('α,'β) *packet* ↦ *unit*)
**where**
*deny-all-from-port-prot p src-net s-port* =
{*pa*. *dest-protocol pa* = *p*} ◁ *deny-all-from-port src-net s-port*

**definition**
*allow-all-to-port-prot* :: *protocol* => 'α::*adr net* ⇒ 'γ::*port* ⇒ (('α,'β) *packet* ↦ *unit*) **where**
*allow-all-to-port-prot p dest-net d-port* =
{*pa*. *dest-protocol pa* = *p*} ◁ *allow-all-to-port dest-net d-port*

**definition**
*deny-all-to-port-prot* :: *protocol* => 'α::*adr net* ⇒ 'γ::*port* ⇒ (('α,'β) *packet* ↦ *unit*) **where**
*deny-all-to-port-prot p dest-net d-port* =
{*pa*. *dest-protocol pa* = *p*} ◁ *deny-all-to-port dest-net d-port*

**definition**
*allow-all-from-port-to-prot*:: *protocol* => 'α::*adr net* ⇒ 'γ::*port* ⇒ 'α::*adr net* ⇒ (('α,'β) *packet* ↦ *unit*)
**where**
*allow-all-from-port-to-prot p src-net s-port dest-net* =
{*pa*. *dest-protocol pa* = *p*} ◁ *allow-all-from-port-to src-net s-port dest-net*

**definition**
*deny-all-from-port-to-prot*::*protocol* ⇒ 'α::*adr net* ⇒ 'γ::*port* ⇒ 'α::*adr net* ⇒ (('α,'β) *packet* ↦ *unit*)
**where**
*deny-all-from-port-to-prot p src-net s-port dest-net* =
{*pa*. *dest-protocol pa* = *p*} ◁ *deny-all-from-port-to src-net s-port dest-net*

**definition**
*allow-all-from-port-to-port-prot*::*protocol* ⇒ 'α::*adr net* ⇒ 'γ::*port* ⇒ 'α::*adr net* ⇒ 'γ::*port* ⇒
(('α,'β) *packet* ↦ *unit*) **where**
*allow-all-from-port-to-port-prot p src-net s-port dest-net d-port* =
{*pa*. *dest-protocol pa* = *p*} ◁ *allow-all-from-port-to-port src-net s-port dest-net d-port*

**definition**
*deny-all-from-port-to-port-prot* :: *protocol* => 'α::*adr net* ⇒ 'γ::*port* ⇒ 'α::*adr net* ⇒
'γ::*port* ⇒ (('α,'β) *packet* ↦ *unit*) **where**
*deny-all-from-port-to-port-prot p src-net s-port dest-net d-port* =
{*pa*. *dest-protocol pa* = *p*} ◁ *deny-all-from-port-to-port src-net s-port dest-net d-port*

**definition**
*allow-all-from-to-port-prot* :: *protocol* => 'α::*adr net* ⇒ 'α::*adr net* ⇒
'γ::*port* ⇒ (('α,'β) *packet* ↦ *unit*) **where**
*allow-all-from-to-port-prot p src-net dest-net d-port* =
{*pa*. *dest-protocol pa* = *p*} ◁ *allow-all-from-to-port src-net dest-net d-port*

**definition**

*deny-all-from-to-port-prot* :: *protocol* =>'α::*adr net* ⇒ 'α::*adr net* ⇒ 'γ::*port* ⇒
(('α,'β) *packet* ↦ *unit*) **where**
*deny-all-from-to-port-prot p src-net dest-net d-port* =
{*pa*. *dest-protocol pa* = *p*} ◁ *deny-all-from-to-port src-net dest-net d-port*

**definition**
*allow-from-port-to-prot* :: *protocol* =>'γ::*port* ⇒ 'α::*adr net* ⇒ 'α::*adr net* ⇒ (('α,'β) *packet* ↦ *unit*)
**where**
*allow-from-port-to-prot p port src-net dest-net* =
{*pa*. *dest-protocol pa* = *p*} ◁ *allow-from-port-to port src-net dest-net*

**definition**
*deny-from-port-to-prot* :: *protocol* =>'γ::*port* ⇒ 'α::*adr net* ⇒ 'α::*adr net* ⇒ (('α,'β) *packet* ↦ *unit*)
**where**
*deny-from-port-to-prot p port src-net dest-net* =
{*pa*. *dest-protocol pa* = *p*} ◁ *deny-from-port-to port src-net dest-net*

**definition**
*allow-from-to-port-prot* :: *protocol* =>'γ::*port* ⇒ 'α::*adr net* ⇒ 'α::*adr net* ⇒ (('α,'β) *packet* ↦ *unit*)
**where**
*allow-from-to-port-prot p port src-net dest-net* =
{*pa*. *dest-protocol pa* = *p*} ◁ *allow-from-to-port port src-net dest-net*

**definition**
*deny-from-to-port-prot* :: *protocol* =>'γ::*port* ⇒ 'α::*adr net* ⇒ 'α::*adr net* ⇒ (('α,'β) *packet* ↦ *unit*)
**where**
*deny-from-to-port-prot p port src-net dest-net* =
{*pa*. *dest-protocol pa* = *p*} ◁ *deny-from-to-port port src-net dest-net*

**definition**
*allow-from-ports-to-prot* :: *protocol* =>'γ::*port set* ⇒ 'α::*adr net* ⇒ 'α::*adr net* ⇒
(('α,'β) *packet* ↦ *unit*) **where**
*allow-from-ports-to-prot p ports src-net dest-net* =
{*pa*. *dest-protocol pa* = *p*} ◁ *allow-from-ports-to ports src-net dest-net*

**definition**
*allow-from-to-ports-prot* :: *protocol* =>'γ::*port set* ⇒ 'α::*adr net* ⇒ 'α::*adr net* ⇒
(('α,'β) *packet* ↦ *unit*) **where**
*allow-from-to-ports-prot p ports src-net dest-net* =
{*pa*. *dest-protocol pa* = *p*} ◁ *allow-from-to-ports ports src-net dest-net*

**definition**
*deny-from-ports-to-prot* :: *protocol* =>'γ::*port set* ⇒ 'α::*adr net* ⇒ 'α::*adr net* ⇒
(('α,'β) *packet* ↦ *unit*) **where**

*deny-from-ports-to-prot p ports src-net dest-net =*
*{pa. dest-protocol pa = p} ◁ deny-from-ports-to ports src-net dest-net*

**definition**
*deny-from-to-ports-prot* :: *protocol* => *'γ*::*port set* ⇒ *'α*::*adr net* ⇒ *'α*::*adr net* ⇒
*(('α,'β) packet ↦ unit)* **where**
*deny-from-to-ports-prot p ports src-net dest-net =*
*{pa. dest-protocol pa = p} ◁ deny-from-to-ports ports src-net dest-net*

As before, we put all the rules into one lemma to ease writing later.

**lemmas** *ProtocolCombinatorsCore =*
*allow-all-from-port-prot-def deny-all-from-port-prot-def allow-all-to-port-prot-def*
*deny-all-to-port-prot-def allow-all-from-to-port-prot-def*
*deny-all-from-to-port-prot-def*
*allow-from-ports-to-prot-def allow-from-to-ports-prot-def*
*deny-from-ports-to-prot-def deny-from-to-ports-prot-def*
*allow-all-from-port-to-prot-def deny-all-from-port-to-prot-def*
*allow-from-port-to-prot-def allow-from-to-port-prot-def deny-from-to-port-prot-def*
*deny-from-port-to-prot-def*

**lemmas** *ProtocolCombinators = PortCombinators.PortCombinators*
*ProtocolCombinatorsCore*

**end**

## 6.5. Ports

**theory** *Ports*
**imports** *Main*
**begin**

This theory can be used if we want to specify the port numbers by names denoting their default Integer values. If you want to use them, please add *Ports* to the simplifier before test data generation.

**definition** *http*::*int* **where** *http = 80*

**lemma** *http1*: *x ≠ 80 ⟹ x ≠ http*
⟨*proof*⟩

**lemma** *http2*: *x ≠ 80 ⟹ http ≠ x*
⟨*proof*⟩

**definition** *smtp*::*int* **where** *smtp = 25*

**lemma** *smtp1*: $x \neq 25 \Longrightarrow x \neq smtp$
⟨*proof*⟩

**lemma** *smtp2*: $x \neq 25 \Longrightarrow smtp \neq x$
⟨*proof*⟩

**definition** *ftp*::*int* **where** *ftp = 21*

**lemma** *ftp1*: $x \neq 21 \Longrightarrow x \neq ftp$
⟨*proof*⟩

**lemma** *ftp2*: $x \neq 21 \Longrightarrow ftp \neq x$
⟨*proof*⟩

And so on for all desired port numbers.

**lemmas** *Ports = http1 http2 ftp1 ftp2 smtp1 smtp2*

**end**

# 7. Network Address Translation

**theory** *NAT*
**imports** *../PacketFilter/PacketFilter*
**begin**

**definition** *src2pool* :: $'\alpha$ *set* ⇒ ($'\alpha$::*adr*,$'\beta$) *packet* ⇒ ($'\alpha$,$'\beta$) *packet set* **where**
*src2pool t* = (λ *p*. ({(*i*,*s*,*d*,*da*). (*i = id p* ∧ *s* ∈ *t* ∧ *d = dest p* ∧ *da = content p*)}))

**definition** *src2poolAP* **where**
*src2poolAP t* = $A_f$ (*src2pool t*)

**definition** *srcNat2pool* :: $'\alpha$ *set* ⇒ $'\alpha$ *set* ⇒ ($'\alpha$::*adr*,$'\beta$) *packet* ↦ ($'\alpha$,$'\beta$) *packet set* **where**
*srcNat2pool srcs transl* = {*x*. *src x* ∈ *srcs*} ◁ (*src2poolAP transl*)

**definition** *src2poolPort* :: *int set* ⇒ ($adr_{ip}$,$'\beta$) *packet* ⇒ ($adr_{ip}$,$'\beta$) *packet set* **where**
*src2poolPort t* = (λ *p*. ({(*i*,(*s1*,*s2*),(*d1*,*d2*),*da*).
(*i = id p* ∧ *s1* ∈ *t* ∧ *s2* = (*snd* (*src p*)) ∧ *d1* = (*fst* (*dest p*)) ∧
*d2* = *snd* (*dest p*) ∧ *da = content p*)}))

**definition** *src2poolPort-Protocol* :: *int set* ⇒ ($adr_{ipp}$,$'\beta$) *packet* ⇒ ($adr_{ipp}$,$'\beta$) *packet set* **where**

*src2poolPort-Protocol t* = ($\lambda$ *p*. ({(*i*,(*s1*,*s2*,*s3*),(*d1*,*d2*,*d3*), *da*).
(*i* = *id p* $\wedge$ *s1* $\in$ *t* $\wedge$ *s2* = (*fst* (*snd* (*src p*))) $\wedge$ *s3* = *snd* (*snd* (*src p*)) $\wedge$
(*d1*,*d2*,*d3*) = *dest p* $\wedge$ *da* = *content p*)}))

**definition** *srcNat2pool-IntPort* :: *address set* $\Rightarrow$ *address set* $\Rightarrow$
($adr_{ip}$,'$\beta$) *packet* $\mapsto$ ($adr_{ip}$,'$\beta$) *packet set* **where**
*srcNat2pool-IntPort srcs transl* =
{*x*. *fst* (*src x*) $\in$ *srcs*} $\triangleleft$ ($A_f$ (*src2poolPort transl*))

**definition** *srcNat2pool-IntProtocolPort* :: *int set* $\Rightarrow$ *int set* $\Rightarrow$
($adr_{ipp}$,'$\beta$) *packet* $\mapsto$ ($adr_{ipp}$,'$\beta$) *packet set* **where**
*srcNat2pool-IntProtocolPort srcs transl* =
{*x*. (*fst* ( (*src x*))) $\in$ *srcs*} $\triangleleft$ ($A_f$ (*src2poolPort-Protocol transl*))

**definition** *srcPat2poolPort-t* :: *int set* $\Rightarrow$ ($adr_{ip}$,'$\beta$) *packet* $\Rightarrow$ ($adr_{ip}$,'$\beta$) *packet set* **where**
*srcPat2poolPort-t t* = ($\lambda$ *p*. ({(*i*,(*s1*,*s2*),(*d1*,*d2*),*da*).
(*i* = *id p* $\wedge$ *s1* $\in$ *t* $\wedge$ *d1* = (*fst* (*dest p*)) $\wedge$ *d2* = *snd* (*dest p*)$\wedge$ *da* = *content p*)}))

**definition** *srcPat2poolPort-Protocol-t* :: *int set* $\Rightarrow$ ($adr_{ipp}$,'$\beta$) *packet* $\Rightarrow$ ($adr_{ipp}$,'$\beta$) *packet set* **where**
*srcPat2poolPort-Protocol-t t* = ($\lambda$ *p*. ({(*i*,(*s1*,*s2*,*s3*),(*d1*,*d2*,*d3*),*da*).
(*i* = *id p* $\wedge$ *s1* $\in$ *t* $\wedge$ *s3* = *src-protocol p* $\wedge$ (*d1*,*d2*,*d3*) = *dest p* $\wedge$ *da* = *content p*)}))

**definition** *srcPat2pool-IntPort* :: *int set* $\Rightarrow$ *int set* $\Rightarrow$ ($adr_{ip}$,'$\beta$) *packet* $\mapsto$
($adr_{ip}$,'$\beta$) *packet set* **where**
*srcPat2pool-IntPort srcs transl* =
{*x*. (*fst* (*src x*)) $\in$ *srcs*} $\triangleleft$ ($A_f$ (*srcPat2poolPort-t transl*))

**definition** *srcPat2pool-IntProtocol* ::
*int set* $\Rightarrow$ *int set* $\Rightarrow$ ($adr_{ipp}$,'$\beta$) *packet* $\mapsto$ ($adr_{ipp}$,'$\beta$) *packet set* **where**

*srcPat2pool-IntProtocol srcs transl* =
{*x*. (*fst* (*src x*)) $\in$ *srcs*} $\triangleleft$ ($A_f$ (*srcPat2poolPort-Protocol-t transl*))

The following lemmas are used for achieving a "standard" output format of NATted packets for use in a firewall execution tool.

**lemma** *datasimp*: {(*i*, (*s1*, *s2*, *s3*), *aba*).
$\forall$ *a aa b ba*. *aba* = ((*a*, *aa*, *b*), *ba*) $\longrightarrow$ *i* = *i1* $\wedge$ *s1* = *i101* $\wedge$
*s3* = *iudp* $\wedge$ *a* = *i110* $\wedge$ *aa* = *X606X3* $\wedge$ *b* = *X607X4* $\wedge$ *ba* = *data*} =
{(*i*, (*s1*, *s2*, *s3*), *aba*).
*i* = *i1* $\wedge$ *s1* = *i101* $\wedge$ *s3* = *iudp* $\wedge$ ($\lambda$ ((*a*,*aa*,*b*),*ba*). *a* = *i110* $\wedge$ *aa* = *X606X3* $\wedge$
*b* = *X607X4* $\wedge$ *ba* = *data*) *aba*}
⟨*proof*⟩

**lemma** *datasimp2*: {(*i*, (*s1*, *s2*, *s3*), *aba*).
$\forall\, a\ aa\ b\ ba.\ aba = ((a,\ aa,\ b),\ ba) \longrightarrow i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge$
$s2 = i1 \wedge a = i110 \wedge aa = i4 \wedge b = iudp \wedge ba = data\}$
$= \{(i,\ (s1,\ s2,\ s3),\ aba).$
$i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge s2 = i1 \wedge (\lambda\ ((a,aa,b),ba).\ a = i110 \wedge$
$aa = i4 \wedge b = iudp \wedge ba = data)\ aba\}$
⟨*proof*⟩

**lemma** *datasimp3*: {(*i*, (*s1*, *s2*, *s3*), *aba*).
*ALL a aa b ba. aba* = ((*a*, *aa*, *b*), *ba*) −−> *i* = *i1* & *i115* < *s1* & *s1* < *i124* &
*s3* = *iudp* & *s2* = *ii1* & *a* = *i110* & *aa* = *i3* & *b* = *itcp* & *ba* = *data*}=
{(*i*, (*s1*, *s2*, *s3*), *aba*).
*i* = *i1* & *i115* < *s1* & *s1* < *i124* & *s3* = *iudp* & *s2* = *ii1* & (λ ((*a*,*aa*,*b*),*ba*).
*a* = *i110* & *aa* = *i3* & *b* = *itcp* & *ba* = *data*) *aba*}
⟨*proof*⟩

**lemma** *datasimp4*: {(*i*, (*s1*, *s2*, *s3*), *aba*).
$\forall\, a\ aa\ b\ ba.\ aba = ((a,\ aa,\ b),\ ba) \longrightarrow i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge$
$s2 = ii1 \wedge a = i110 \wedge aa = i7 \wedge b = itcp \wedge ba = data\}=$
$\{(i,\ (s1,\ s2,\ s3),\ aba).$
$i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge s2 = ii1 \wedge (\lambda\ ((a,aa,b),ba).\ a = i110 \wedge$
$aa = i7 \wedge b = itcp \wedge ba = data)\ aba\}$
⟨*proof*⟩

**lemma** *datasimp5*: {(*i*, (*s1*, *s2*, *s3*), *aba*).
*i* = *i1* ∧ *s1* = *i101* ∧ *s3* = *iudp* ∧ (λ ((*a*,*aa*,*b*),*ba*). *a* = *i110* ∧ *aa* = *X606X3* ∧
*b* = *X607X4* ∧ *ba* = *data*) *aba*}
= {(*i*, (*s1*, *s2*, *s3*), (*a*,*aa*,*b*),*ba*).
*i* = *i1* ∧ *s1* = *i101* ∧ *s3* = *iudp* ∧ *a* = *i110* ∧ *aa* = *X606X3* ∧
*b* = *X607X4* ∧ *ba* = *data*}
⟨*proof*⟩

**lemma** *datasimp6*: {(*i*, (*s1*, *s2*, *s3*), *aba*).
*i* = *i1* ∧ *s1* = *i132* ∧ *s3* = *iudp* ∧ *s2* = *i1* ∧ (λ ((*a*,*aa*,*b*),*ba*). *a* = *i110* ∧
*aa* = *i4* ∧ *b* = *iudp* ∧ *ba* = *data*) *aba*}=
{(*i*, (*s1*, *s2*, *s3*), (*a*,*aa*,*b*),*ba*).
*i* = *i1* ∧ *s1* = *i132* ∧ *s3* = *iudp* ∧ *s2* = *i1* ∧ *a* = *i110* ∧ *aa* = *i4* ∧ *b* = *iudp* ∧ *ba*
= *data*}
⟨*proof*⟩

**lemma** *datasimp7*: {(*i*, (*s1*, *s2*, *s3*), *aba*).
*i* = *i1* & *i115* < *s1* & *s1* < *i124* & *s3* = *iudp* & *s2* = *ii1* & (λ ((*a*,*aa*,*b*),*ba*). *a* =
*i110* &
*aa* = *i3* & *b* = *itcp* & *ba* = *data*) *aba*} =
{(*i*, (*s1*, *s2*, *s3*), (*a*,*aa*,*b*),*ba*).
*i* = *i1* & *i115* < *s1* & *s1* < *i124* & *s3* = *iudp* & *s2* = *ii1* & *a* = *i110* & *aa* = *i3* & *b* = *itcp*
& *ba* = *data*}
⟨*proof*⟩

**lemma** *datasimp8*: {(*i*, (*s1*, *s2*, *s3*), *aba*).
*i* = *i1* ∧ *s1* = *i132* ∧ *s3* = *iudp* ∧ *s2* = *ii1* ∧ (λ ((*a*,*aa*,*b*),*ba*). *a* = *i110* ∧ *aa* = *i7* ∧
*b* = *itcp* ∧ *ba* = *data*) *aba*}
= {(*i*, (*s1*, *s2*, *s3*), (*a*,*aa*,*b*),*ba*).
*i* = *i1* ∧ *s1* = *i132* ∧ *s3* = *iudp* ∧ *s2* = *ii1* ∧ *a* = *i110* ∧ *aa* = *i7* ∧ *b* = *itcp* ∧
*ba* = *data*}
⟨*proof*⟩

**lemmas** *datasimps* = *datasimp datasimp2 datasimp3 datasimp4*
*datasimp5 datasimp6 datasimp7 datasimp8*

**lemmas** *NATLemmas* = *src2pool-def src2poolPort-def*
*src2poolPort-Protocol-def src2poolAP-def srcNat2pool-def*
*srcNat2pool-IntProtocolPort-def srcNat2pool-IntPort-def*
*srcPat2poolPort-t-def srcPat2poolPort-Protocol-t-def*
*srcPat2pool-IntPort-def srcPat2pool-IntProtocol-def*

**end**

# 8. Policy Normalisation

**theory**
*FWNormalisationCore*
**imports**
*../PacketFilter/PacketFilter*
**begin**

This theory contains all the definitions used for policy normalisation as described in [1]. The normalisation procedure transforms policies into semantically equivalent ones which are "easier" to test. It is organized into nine phases. We impose the following two restrictions on the input policies:

- Each policy must contain a `DenyAll` rule. If this restriction were to be lifted, the

`insertDenies` phase would have to be adjusted accordingly.

- For each pair of networks $n_1$ and $n_2$, the networks are either disjoint or equal. If this restriction were to be lifted, we would need some additional phases before the start of the normalisation procedure presented below. This rule would split single rules into several by splitting up the networks such that they are all pairwise disjoint or equal. Such a transformation is clearly semantics-preserving and the condition would hold after these phases.

As a result, the procedure generates a list of policies, in which:

- each element of the list contains a policy which completely specifies the blocking behavior between two networks, and
- there are no shadowed rules.

This result is desirable since the test case generation for rules between networks $A$ and $B$ is independent of the rules that specify the behavior for traffic flowing between networks $C$ and $D$. Thus, the different segments of the policy can be processed individually. The normalization procedure does not aim to minimize the number of rules. While it does remove unnecessary ones, it also adds new ones, enabling a policy to be split into several independent parts.

Policy transformations are functions that map policies to policies. We decided to represent policy transformations as *syntactic rules*; this choice paves the way for expressing the entire normalisation process inside HOL by functions manipulating abstract policy syntax.

## 8.1. Basics

We define a very simple policy language:

**datatype** $('\alpha,'\beta)$ *Combinators* =
  *DenyAll*
| *DenyAllFromTo* $'\alpha$ $'\alpha$
| *AllowPortFromTo* $'\alpha$ $'\alpha$ $'\beta$
| *Conc* $(('\alpha,'\beta)$ *Combinators*$)$ $(('\alpha,'\beta)$ *Combinators*$)$ (**infixr** $\oplus$ *80*)

And define the semantic interpretation of it. For technical reasons, we fix here the type to policies over IntegerPort addresses. However, we could easily provide definitions for other address types as well, using a generic consts for the type definition and a primrec definition for each desired address model.

## 8.2. Auxiliary definitions and functions.

This subsection defines several functions which are useful later for the combinators, invariants, and proofs.

**fun** *srcNet* **where**
 *srcNet* (*DenyAllFromTo x y*) = *x*
|*srcNet* (*AllowPortFromTo x y p*) = *x*

```
|srcNet DenyAll = undefined
|srcNet (v ⊕ va) = undefined

fun destNet where
 destNet (DenyAllFromTo x y) = y
|destNet (AllowPortFromTo x y p) = y
|destNet DenyAll = undefined
|destNet (v ⊕ va) = undefined


fun srcnets where
 srcnets DenyAll = []
|srcnets (DenyAllFromTo x y) = [x]
|srcnets (AllowPortFromTo x y p) = [x]
|(srcnets (x ⊕ y)) = (srcnets x)@(srcnets y)

fun destnets where
 destnets DenyAll = []
|destnets (DenyAllFromTo x y) = [y]
|destnets (AllowPortFromTo x y p) = [y]
|(destnets (x ⊕ y)) = (destnets x)@(destnets y)

fun (sequential) net-list-aux where
 net-list-aux [] = []
|net-list-aux (DenyAll#xs) = net-list-aux xs
|net-list-aux ((DenyAllFromTo x y)#xs) = x#y#(net-list-aux xs)
|net-list-aux ((AllowPortFromTo x y p)#xs) = x#y#(net-list-aux xs)
|net-list-aux ((x⊕y)#xs) = (net-list-aux [x])@(net-list-aux [y])@(net-list-aux xs)

fun net-list where net-list p = remdups (net-list-aux p)

definition bothNets where bothNets x = (zip (srcnets x) (destnets x))

fun (sequential) normBothNets where
 normBothNets ((a,b)#xs) = (if ((b,a) ∈ set xs) ∨ (a,b) ∈ set (xs)
                            then (normBothNets xs)
                            else (a,b)#(normBothNets xs))
|normBothNets x = x

fun makeSets where
 makeSets ((a,b)#xs) = ({a,b}#(makeSets xs))
|makeSets [] = []

fun bothNet where
 bothNet DenyAll = {}
|bothNet (DenyAllFromTo a b) = {a,b}
|bothNet (AllowPortFromTo a b p) = {a,b}
|bothNet (v ⊕ va) = undefined
```

*Nets_List* provides from a list of rules a list where the entries are the appearing sets of

source and destination network of each rule.

**definition** *Nets-List* **where** *Nets-List x = makeSets (normBothNets (bothNets x))*

**fun** (*sequential*) *first-srcNet* **where**
*first-srcNet (x⊕y) = first-srcNet x*
| *first-srcNet x = srcNet x*

**fun** (*sequential*) *first-destNet* **where**
*first-destNet (x⊕y) = first-destNet x*
| *first-destNet x = destNet x*

**fun** (*sequential*) *first-bothNet* **where**
*first-bothNet (x⊕y) = first-bothNet x*
|*first-bothNet x = bothNet x*

**fun** (*sequential*) *in-list* **where**
*in-list DenyAll l = True*
|*in-list x l = (bothNet x ∈ set l)*

**fun** *all-in-list* **where**
*all-in-list [] l = True*
|*all-in-list (x#xs) l = (in-list x l ∧ all-in-list xs l)*

**fun** (*sequential*) *member* **where**
*member a (x⊕xs) = ((member a x) ∨ (member a xs))*
|*member a x = (a = x)*

**fun** *sdnets* **where**
*sdnets DenyAll = {}*
| *sdnets (DenyAllFromTo a b) = {(a,b)}*
| *sdnets (AllowPortFromTo a b c) = {(a,b)}*
| *sdnets (a ⊕ b) = sdnets a ∪ sdnets b*

**definition** *packet-Nets* **where** *packet-Nets x a b = ((src x ⊏ a ∧ dest x ⊏ b) ∨ (src x ⊏ b ∧ dest x ⊏ a))*

**definition** *subnetsOfAdr* **where** *subnetsOfAdr a = {x. a ⊏ x}*

**definition** *fst-set* **where** *fst-set s = {a. ∃ b. (a,b) ∈ s}*

**definition** *snd-set* **where** *snd-set s = {a. ∃ b. (b,a) ∈ s}*

**fun** *memberP* **where**
*memberP r (x#xs) = (member r x ∨ memberP r xs)*
|*memberP r [] = False*

**fun** *firstList* **where**
*firstList (x#xs) = (first-bothNet x)*

$|$*firstList* [] = {}

## 8.3. Invariants

If there is a DenyAll, it is at the first position

**fun** *wellformed-policy1*:: ((′$\alpha$, ′$\beta$) *Combinators*) *list* ⇒ *bool* **where**
  *wellformed-policy1* [] = *True*
| *wellformed-policy1* (*x*#*xs*) = (*DenyAll* ∉ (*set xs*))

There is a DenyAll at the first position

**fun** *wellformed-policy1-strong*:: ((′$\alpha$, ′$\beta$) *Combinators*) *list* ⇒ *bool*
**where**
  *wellformed-policy1-strong* [] = *False*
| *wellformed-policy1-strong* (*x*#*xs*) = (*x*=*DenyAll* ∧ (*DenyAll* ∉ (*set xs*)))

All two networks are either disjoint or equal.

**definition** *netsDistinct* **where** *netsDistinct a b* = (¬ (∃ *x*. *x* ⊏ *a* ∧ *x* ⊏ *b*))

**definition** *twoNetsDistinct* **where**
  *twoNetsDistinct a b c d* = (*netsDistinct a c* ∨ *netsDistinct b d*)

**definition** *allNetsDistinct* **where**
  *allNetsDistinct p* = (∀ *a b*. (*a* ≠ *b* ∧ *a* ∈ *set* (*net-list p*) ∧
                    *b* ∈ *set* (*net-list p*)) ⟶ *netsDistinct a b*)

**definition** *disjSD-2* **where**
 *disjSD-2 x y* = (∀ *a b c d*. ((*a*,*b*)∈*sdnets x* ∧ (*c*,*d*) ∈*sdnets y* ⟶
     (*twoNetsDistinct a b c d* ∧ *twoNetsDistinct a b d c*)))

The policy is given as a list of single rules.

**fun** *singleCombinators* **where**
*singleCombinators* [] = *True*
|*singleCombinators* ((*x*⊕*y*)#*xs*) = *False*
|*singleCombinators* (*x*#*xs*) = *singleCombinators xs*

**definition** *onlyTwoNets* **where**
 *onlyTwoNets x* = ((∃ *a b*. (*sdnets x* = {(*a*,*b*)})) ∨ (∃ *a b*. *sdnets x* = {(*a*,*b*),(*b*,*a*)}))

Each entry of the list contains rules between two networks only.

**fun** *OnlyTwoNets* **where**
 *OnlyTwoNets* (*DenyAll*#*xs*) = *OnlyTwoNets xs*
|*OnlyTwoNets* (*x*#*xs*) = (*onlyTwoNets x* ∧ *OnlyTwoNets xs*)
|*OnlyTwoNets* [] = *True*

**fun** *noDenyAll* **where**
 *noDenyAll* (*x*#*xs*) = ((¬ *member DenyAll x*) ∧ *noDenyAll xs*)
|*noDenyAll* [] = *True*

**fun** *noDenyAll1* **where**
  *noDenyAll1* (*DenyAll*#*xs*) = *noDenyAll xs*
| *noDenyAll1 xs* = *noDenyAll xs*

**fun** *separated* **where**
  *separated* (*x*#*xs*) = ((∀ *s*. *s* ∈ *set xs* ⟶ *disjSD-2 x s*) ∧ *separated xs*)
| *separated* [] = *True*

**fun** *NetsCollected* **where**
  *NetsCollected* (*x*#*xs*) = (((*first-bothNet x* ≠ *firstList xs*) ⟶
        (∀ *a*∈*set xs*. *first-bothNet x* ≠ *first-bothNet a*)) ∧ *NetsCollected* (*xs*))
| *NetsCollected* [] = *True*

**fun** *NetsCollected2* **where**
 *NetsCollected2* (*x*#*xs*) = (*xs* = [] ∨ (*first-bothNet x* ≠ *firstList xs* ∧
                    *NetsCollected2 xs*))
|*NetsCollected2* [] = *True*

## 8.4. Transformations

The following two functions transform a policy into a list of single rules and vice-versa - by staying on the combinator level.

**fun** *policy2list*::('α, 'β) *Combinators* ⇒
            (('α, 'β) *Combinators*) *list* **where**
 *policy2list* (*x* ⊕ *y*) = (*concat* [(*policy2list x*),(*policy2list y*)])
|*policy2list x* = [*x*]

**fun** *list2FWpolicy*::((('α, 'β) *Combinators*) *list* ⇒
              (('α, 'β) *Combinators*) **where**
 *list2FWpolicy* [] = *undefined*
|*list2FWpolicy* (*x*#[]) = *x*
|*list2FWpolicy* (*x*#*y*) = *x* ⊕ (*list2FWpolicy y*)

Remove all the rules appearing before a DenyAll. There are two alternative versions.

**fun** *removeShadowRules1* **where**
  *removeShadowRules1* (*x*#*xs*) = (*if* (*DenyAll* ∈ *set xs*)
                    *then* ((*removeShadowRules1 xs*))
                    *else x*#*xs*)
| *removeShadowRules1* [] = []

**fun** *removeShadowRules1-alternative-rev* **where**
  *removeShadowRules1-alternative-rev* [] = []
| *removeShadowRules1-alternative-rev* (*DenyAll*#*xs*) = [*DenyAll*]
| *removeShadowRules1-alternative-rev* [*x*] = [*x*]
| *removeShadowRules1-alternative-rev* (*x*#*xs*)=
          *x*#(*removeShadowRules1-alternative-rev xs*)

**definition** *removeShadowRules1-alternative* **where**

```
  removeShadowRules1-alternative p =
                    rev (removeShadowRules1-alternative-rev (rev p))
```

Remove all the rules which allow a port, but are shadowed by a deny between these subnets

```
fun removeShadowRules2:: (('α, 'β) Combinators) list ⇒
                         (('α, 'β) Combinators) list
where
  (removeShadowRules2 ((AllowPortFromTo x y p)#z)) =
       (if (((DenyAllFromTo x y) ∈ set z))
        then ((removeShadowRules2 z))
        else (((AllowPortFromTo x y p)#(removeShadowRules2 z))))
| removeShadowRules2 (x#y) = x#(removeShadowRules2 y)
| removeShadowRules2 [] = []
```

Sorting a pocliy. We first need to define an ordering on rules. This ordering depends on the $Nets\_List$ of a policy.

```
fun smaller :: ('α, 'β) Combinators ⇒
               ('α, 'β) Combinators ⇒
               (('α) set) list ⇒ bool
where
 smaller DenyAll x l = True
| smaller x DenyAll l = False
| smaller x y l =
 ((x = y) ∨          (if (bothNet x) = (bothNet y) then
                      (case y of (DenyAllFromTo a b) ⇒ (x = DenyAllFromTo b a)
                          | - ⇒  True)
                  else
                     (position (bothNet x) l <= position (bothNet y) l)))
```

We provide two different sorting algorithms: Quick Sort (qsort) and Insertion Sort (sort)

```
fun qsort where
  qsort [] l     = []
| qsort (x#xs) l = (qsort [y←xs. ¬ (smaller x y l)] l) @ [x] @ (qsort [y←xs. smaller x y l] l)

lemma qsort-permutes:
  set (qsort xs l) = set xs
  ⟨proof⟩

lemma set-qsort [simp]: set (qsort xs l)  = set xs
 ⟨proof⟩


fun insort where
   insort a [] l = [a]
 | insort a (x#xs) l = (if (smaller a x l) then a#x#xs else x#(insort a xs l))

fun sort where
```

```
  sort [] l = []
| sort (x#xs) l = insort x (sort xs l) l
```

```
fun sorted where
sorted [] l ⟷ True |
sorted [x] l ⟷ True |
sorted (x#y#zs) l ⟷ smaller x y l ∧ sorted (y#zs) l
```

```
fun separate where
 separate (DenyAll#x) = DenyAll#(separate x)
| separate (x#y#z) = (if (first-bothNet x = first-bothNet y)
                  then (separate ((x⊕y)#z))
                  else (x#(separate(y#z))))
|separate x = x
```

Insert the DenyAllFromTo rules, such that traffic between two networks can be tested individually

```
fun  insertDenies where
 insertDenies (x#xs) = (case x of DenyAll ⇒ (DenyAll#(insertDenies xs))
                    | - ⇒ (DenyAllFromTo (first-srcNet x) (first-destNet x) ⊕
                       (DenyAllFromTo (first-destNet x) (first-srcNet x)) ⊕ x)#
                             (insertDenies xs))
| insertDenies [] = []
```

Remove duplicate rules. This is especially necessary as insertDenies might have inserted duplicate rules.

The second function is supposed to work on a list of policies. Only rules which are duplicated within the same policy are removed.

```
fun removeDuplicates where
  removeDuplicates (x⊕xs) = (if member x xs then (removeDuplicates xs)
                           else x⊕(removeDuplicates xs))
| removeDuplicates x = x
```

```
fun removeAllDuplicates where
 removeAllDuplicates (x#xs) = ((removeDuplicates (x))#(removeAllDuplicates xs))
|removeAllDuplicates x = x
```

Insert a DenyAll at the beginning of a policy.

```
fun insertDeny where
 insertDeny (DenyAll#xs) = DenyAll#xs
|insertDeny xs = DenyAll#xs
```

```
definition sort' p l = sort l p
```

```
definition qsort' p l = qsort l p
```

**declare** *dom-eq-empty-conv* [*simp del*]

**fun** *list2policyR*::((′α, ′β) *Combinators*) *list* ⇒
((′α, ′β) *Combinators*) **where**
*list2policyR* (*x*#[]) = *x*
|*list2policyR* (*x*#*y*) = (*list2policyR y*) ⊕ *x*
|*list2policyR* [] = *undefined*

We provide the definitions for two address representations.

## 8.5. IntPort

**fun** *C* :: (*adr*$_{ip}$ *net*, *port*) *Combinators* ⇒ (*adr*$_{ip}$,*DummyContent*) *packet* ↦ *unit*
**where**
*C DenyAll* = *deny-all*
|*C* (*DenyAllFromTo x y*) = *deny-all-from-to x y*
|*C* (*AllowPortFromTo x y p*) = *allow-from-to-port p x y*
|*C* (*x* ⊕ *y*) = *C x* ++ *C y*

**fun** *CRotate* :: (*adr*$_{ip}$ *net*, *port*) *Combinators* ⇒ (*adr*$_{ip}$,*DummyContent*) *packet* ↦ *unit*
**where**
*CRotate DenyAll* = *C DenyAll*
|*CRotate* (*DenyAllFromTo x y*) = *C* (*DenyAllFromTo x y*)
|*CRotate* (*AllowPortFromTo x y p*) = *C* (*AllowPortFromTo x y p*)
|*CRotate* (*x* ⊕ *y*) = ((*CRotate y*) ++ ((*CRotate x*)))

**fun** *rotatePolicy* **where**
*rotatePolicy DenyAll* = *DenyAll*
| *rotatePolicy* (*DenyAllFromTo a b*) = *DenyAllFromTo a b*
| *rotatePolicy* (*AllowPortFromTo a b p*) = *AllowPortFromTo a b p*
| *rotatePolicy* (*a*⊕*b*) = (*rotatePolicy b*) ⊕ (*rotatePolicy a*)

**lemma** *check*: *rev* (*policy2list* (*rotatePolicy p*)) = *policy2list p*
⟨*proof*⟩

All rules appearing at the left of a DenyAllFromTo, have disjunct domains from it (except DenyAll)

**fun** (*sequential*) *wellformed-policy2* **where**
*wellformed-policy2* [] = *True*
| *wellformed-policy2* (*DenyAll*#*xs*) = *wellformed-policy2 xs*
| *wellformed-policy2* (*x*#*xs*) = ((∀ *c a b*. *c* = *DenyAllFromTo a b* ∧ *c* ∈ *set xs* ⟶
*Map.dom* (*C x*) ∩ *Map.dom* (*C c*) = {}) ∧ *wellformed-policy2 xs*)

An allow rule is disjunct with all rules appearing at the right of it. This invariant is not necessary as it is a consequence from others, but facilitates some proofs.

**fun** (*sequential*) *wellformed-policy3*::((*adr*$_{ip}$ *net*,*port*) *Combinators*) *list* ⇒ *bool* **where**
  *wellformed-policy3* [] = *True*
| *wellformed-policy3* ((*AllowPortFromTo a b p*)#*xs*) = ((∀ *r*. *r* ∈ *set xs* ⟶
    *dom* (*C r*) ∩ *dom* (*C* (*AllowPortFromTo a b p*)) = {}) ∧ *wellformed-policy3 xs*)
| *wellformed-policy3* (*x*#*xs*) = *wellformed-policy3 xs*

**definition**
  *normalize'* *p* = (*removeAllDuplicates o insertDenies o separate o*
    (*sort'* (*Nets-List p*)) *o removeShadowRules2 o remdups o*
    (*removeShadowRules3 C*) *o insertDeny o removeShadowRules1 o*
    *policy2list*) *p*

**definition**
  *normalizeQ'* *p* = (*removeAllDuplicates o insertDenies o separate o*
    (*qsort'* (*Nets-List p*)) *o removeShadowRules2 o remdups o*
    (*removeShadowRules3 C*) *o insertDeny o removeShadowRules1 o*
    *policy2list*) *p*

**definition** *normalize* ::
  (*adr*$_{ip}$ *net*, *port*) *Combinators* ⇒
  (*adr*$_{ip}$ *net*, *port*) *Combinators list*
**where**
  *normalize p* = (*removeAllDuplicates* (*insertDenies* (*separate* (*sort*
    (*removeShadowRules2* (*remdups* ((*removeShadowRules3 C*) (*insertDeny*
    (*removeShadowRules1* (*policy2list p*)))))) ((*Nets-List p*))))))

**definition**
  *normalize-manual-order p l* = *removeAllDuplicates* (*insertDenies* (*separate*
  (*sort* (*removeShadowRules2* (*remdups* ((*removeShadowRules3 C*) (*insertDeny*
  (*removeShadowRules1* (*policy2list p*)))))) ((*l*)))))

**definition** *normalizeQ* ::
  (*adr*$_{ip}$ *net*, *port*) *Combinators* ⇒
  (*adr*$_{ip}$ *net*, *port*) *Combinators list*
**where**
  *normalizeQ p* = (*removeAllDuplicates* (*insertDenies* (*separate* (*qsort*
    (*removeShadowRules2* (*remdups* ((*removeShadowRules3 C*) (*insertDeny*
    (*removeShadowRules1* (*policy2list p*)))))) ((*Nets-List p*))))))

**definition**
  *normalize-manual-orderQ p l* = *removeAllDuplicates* (*insertDenies* (*separate*
  (*qsort* (*removeShadowRules2* (*remdups* ((*removeShadowRules3 C*) (*insertDeny*

(*removeShadowRules1* (*policy2list p*))))))) (((*l*)))))

Of course, normalize is equal to normalize', the latter looks nicer though.

**lemma** *normalize* = *normalize'*
⟨*proof*⟩

**declare** *C.simps* [*simp del*]

## 8.6. TCP_UDP_IntegerPort

**fun** *Cp* :: (*adr*$_{ipp}$ *net*, *protocol* × *port*) *Combinators* ⇒
(*adr*$_{ipp}$,*DummyContent*) *packet* ↦ *unit*
**where**
*Cp DenyAll* = *deny-all*
|*Cp* (*DenyAllFromTo x y*) = *deny-all-from-to x y*
|*Cp* (*AllowPortFromTo x y p*) = *allow-from-to-port-prot* (*fst p*) (*snd p*) *x y*
|*Cp* (*x* ⊕ *y*) = *Cp x* ++ *Cp y*

**fun** *Dp* :: (*adr*$_{ipp}$ *net*, *protocol* × *port*) *Combinators* ⇒
(*adr*$_{ipp}$,*DummyContent*) *packet* ↦ *unit*
**where**
*Dp DenyAll* = *Cp DenyAll*
|*Dp* (*DenyAllFromTo x y*) = *Cp* (*DenyAllFromTo x y*)
|*Dp* (*AllowPortFromTo x y p*) = *Cp* (*AllowPortFromTo x y p*)
|*Dp* (*x* ⊕ *y*) = *Cp* (*y* ⊕ *x*)

All rules appearing at the left of a DenyAllFromTo, have disjunct domains from it (except DenyAll)

**fun** (*sequential*) *wellformed-policy2Pr* **where**
*wellformed-policy2Pr* [] = *True*
| *wellformed-policy2Pr* (*DenyAll*#*xs*) = *wellformed-policy2Pr xs*
| *wellformed-policy2Pr* (*x*#*xs*) = ((∀ *c a b*. *c* = *DenyAllFromTo a b* ∧ *c* ∈ *set xs* ⟶
*Map.dom* (*Cp x*) ∩ *Map.dom* (*Cp c*) = {}) ∧ *wellformed-policy2Pr xs*)

An allow rule is disjunct with all rules appearing at the right of it. This invariant is not necessary as it is a consequence from others, but facilitates some proofs.

**fun** (*sequential*) *wellformed-policy3Pr*::((*adr*$_{ipp}$ *net*, *protocol* × *port*) *Combinators*) *list* ⇒ *bool*
**where**
*wellformed-policy3Pr* [] = *True*
| *wellformed-policy3Pr* ((*AllowPortFromTo a b p*)#*xs*) = ((∀ *r*. *r* ∈ *set xs* ⟶
*dom* (*Cp r*) ∩ *dom* (*Cp* (*AllowPortFromTo a b p*)) = {}) ∧ *wellformed-policy3Pr xs*)
| *wellformed-policy3Pr* (*x*#*xs*) = *wellformed-policy3Pr xs*

**definition**
*normalizePr'* :: (*adr*$_{ipp}$ *net*, *protocol* × *port*) *Combinators*
⇒ (*adr*$_{ipp}$ *net*, *protocol* × *port*) *Combinators list* **where**
*normalizePr' p* = (*removeAllDuplicates o insertDenies o separate o*
(*sort'* (*Nets-List p*)) *o removeShadowRules2 o remdups o*
(*removeShadowRules3 Cp*) *o insertDeny o removeShadowRules1 o*
*policy2list*) *p*

**definition** *normalizePr* ::
(*adr*$_{ipp}$ *net*, *protocol* × *port*) *Combinators*
⇒ (*adr*$_{ipp}$ *net*, *protocol* × *port*) *Combinators list* **where**
*normalizePr p* = (*removeAllDuplicates* (*insertDenies* (*separate* (*sort*
(*removeShadowRules2* (*remdups* ((*removeShadowRules3 Cp*) (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) ((*Nets-List p*))))))

**definition**
*normalize-manual-orderPr p l* = *removeAllDuplicates* (*insertDenies* (*separate*
(*sort* (*removeShadowRules2* (*remdups* ((*removeShadowRules3 Cp*) (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) ((*l*)))))

**definition**
*normalizePrQ'* :: (*adr*$_{ipp}$ *net*, *protocol* × *port*) *Combinators*
⇒ (*adr*$_{ipp}$ *net*, *protocol* × *port*) *Combinators list* **where**
*normalizePrQ' p* = (*removeAllDuplicates o insertDenies o separate o*
(*qsort'* (*Nets-List p*)) *o removeShadowRules2 o remdups o*
(*removeShadowRules3 Cp*) *o insertDeny o removeShadowRules1 o*
*policy2list*) *p*

**definition** *normalizePrQ* ::
(*adr*$_{ipp}$ *net*, *protocol* × *port*) *Combinators*
⇒ (*adr*$_{ipp}$ *net*, *protocol* × *port*) *Combinators list* **where**
*normalizePrQ p* = (*removeAllDuplicates* (*insertDenies* (*separate* (*qsort*
(*removeShadowRules2* (*remdups* ((*removeShadowRules3 Cp*) (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) ((*Nets-List p*))))))

**definition**
*normalize-manual-orderPrQ p l* = *removeAllDuplicates* (*insertDenies* (*separate*
(*qsort* (*removeShadowRules2* (*remdups* ((*removeShadowRules3 Cp*) (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) ((*l*)))))

Of course, normalize is equal to normalize', the latter looks nicer though.

**lemma** *normalizePr* = *normalizePr'*
⟨*proof*⟩

The following definition helps in creating the test specification for the individual parts of a normalized policy.

**definition** *makeFUTPr* **where**
*makeFUTPr FUT p x n* =
(*packet-Nets x* (*fst* (*normBothNets* (*bothNets p*)!*n*))
(*snd*(*normBothNets* (*bothNets p*)!*n*)) ⟶
*FUT x* = *Cp* ((*normalizePr p*)!*Suc n*) *x*)

**declare** *Cp.simps* [*simp del*]

**lemmas** *PLemmas* = *C.simps Cp.simps dom-def PolicyCombinators.PolicyCombinators PortCombinators.PortCombinatorsCore aux ProtocolPortCombinators.ProtocolCombinatorsCore src-def dest-def in-subnet-def* $adr_{ip}$*Lemmas* $adr_{ipp}$*Lemmas*

**lemma** *aux*: ⟦$x \neq a$; $y \neq b$; ($x \neq y \wedge x \neq b$) ∨ ($a \neq b \wedge a \neq y$)⟧ ⟹ {$x$,$a$} ≠ {$y$,$b$}
⟨*proof*⟩

**lemma** *aux2*: {$a$,$b$} = {$b$,$a$}
⟨*proof*⟩

**end**

# 9. Stateful Firewalls

## 9.1. Basic Constructs

**theory** *Stateful*
**imports** *../PacketFilter/PacketFilter LTL-alike*
**begin**

The simple system of a stateless packet filter is not enough to model all common real-world scenarios. Some protocols need further actions in order to be secured. A prominent example is the File Transfer Protocol (FTP), which is a popular means to move files across the Internet. It behaves quite differently from most other application layer protocols as it uses a two-way connection establishment which opens a dynamic port. A stateless packet filter would only have the possibility to either always open all the

possible dynamic ports or not to allow that protocol at all. Neither of these options is satisfactory. In the first case, all ports above 1024 would have to be opened which introduces a big security hole in the system, in the second case users wouldn't be very happy. A firewall which tracks the state of the TCP connections on a system doesn't help here either, as the opening and closing of the ports takes place on the application layer. Therefore, a firewall needs to have some knowledge of the application protocols being run and track the states of these protocols. We next model this behaviour.

The key point of our model is the idea that a policy remains the same as before: a mapping from packet to packet out. We still specify for every packet, based on its source and destination address, the expected action. The only thing that changes now is that this mapping is allowed to change over time. This indicates that our test data will not consist of single packets but rather of sequences thereof.

At first we hence need a state. It is a tuple from some memory to be refined later and the current policy.

**type-synonym** $('\alpha,'\beta,'\gamma)$ *FWState* $= '\alpha \times (('\beta,'\gamma)$ *packet* $\mapsto$ *unit*$)$

Having a state, we need of course some state transitions. Such a transition can happen every time a new packet arrives. State transitions can be modelled using a state-exception monad. We provide two types of firewall monads: one

**type-synonym** $('\alpha,'\beta,'\gamma)$ *FWStateTransitionP* $=('\beta,'\gamma)$ *packet* $\Rightarrow$
$((('\beta,'\gamma)$ *packet* $\mapsto$ *unit*$)$ *decision*, $('\alpha,'\beta,'\gamma)$ *FWState*$)$ $MON_{SE}$

**type-synonym** $('\alpha,'\beta,'\gamma)$ *FWStateTransition* $=$
$(('\beta,'\gamma)$ *packet* $\times$ $('\alpha,'\beta,'\gamma)$ *FWState*$) \rightharpoonup ('\alpha,'\beta,'\gamma)$ *FWState*

The memory could be modelled as a list of accepted packets.

**type-synonym** $('\beta,'\gamma)$ *history* $=$ $('\beta,'\gamma)$ *packet list*

**fun** *packet-with-id* **where**
*packet-with-id* [] $i$ $=$ []
|*packet-with-id* $(x\#xs)$ $i$ $=$ (*if id* $x = i$ *then* $(x\#$(*packet-with-id xs* $i$)) *else* (*packet-with-id xs* $i$))

**fun** *ids1* **where**
*ids1* $i$ $(x\#xs) = (id\ x = i \wedge ids1\ i\ xs)$
|*ids1* $i$ [] $=$ *True*

**fun** *ids* **where**
*ids* $a$ $(x\#xs) = (NetworkCore.id\ x \in a \wedge ids\ a\ xs)$
|*ids* $a$ [] $=$ *True*

**definition** *applyPolicy*:: $('i \times ('i \mapsto 'o)) \mapsto 'o$
**where** *applyPolicy* $= (\lambda\ (x,z).\ z\ x)$

**end**

## 9.2. FTP Protocol

**theory** *FTP*
**imports**
*Stateful*
**begin**

### 9.2.1. The protocol syntax

The File Transfer Protocol FTP is a well known example of a protocol which uses dynamic ports and is therefore a natural choice to use as an example for our model.

We model only a simplified version of the FTP protocol over IntegerPort addresses, still containing all messages that matter for our purposes. It consists of the following four messages:

1. *init*: The client contacts the server indicating his wish to get some data.
2. *ftp-port-request p*: The client, usually after having received an acknowledgement of the server, indicates a port number on which he wants to receive the data.
3. *ftp-ftp-data*: The server sends the requested data over the new channel. There might be an arbitrary number of such messages, including zero.
4. *ftp-close*: The client closes the connection. The dynamic port gets closed again.

The content field of a packet therefore now consists of either one of those four messages or a default one.

**datatype** *msg = ftp-init | ftp-port-request port | ftp-data | ftp-close | ftp-other*

We now also make use of the ID field of a packet. It is used as session ID and we make the assumption that they are all unique among different protocol runs.

At first, we need some predicates which check if a packet is a specific FTP message and has the correct session ID.

**definition**
*is-init* :: *id* ⇒ (*adr*$_{ip}$, *msg*)*packet* ⇒ *bool* **where**
*is-init* = (λ *i p*. (*id p* = *i* ∧ *content p* = *ftp-init*))

**definition**
*is-ftp-port-request* :: *id* ⇒ *port* ⇒(*adr*$_{ip}$, *msg*) *packet* ⇒ *bool* **where**
*is-ftp-port-request* = (λ *i port p*. (*id p* = *i* ∧ *content p* = *ftp-port-request port*))

**definition**
*is-ftp-data* :: *id* ⇒ (*adr*$_{ip}$, *msg*) *packet* ⇒ *bool* **where**
*is-ftp-data* = (λ *i p*. (*id p* = *i* ∧ *content p* = *ftp-data*))

**definition**
*is-ftp-close* :: *id* ⇒ (*adr*$_{ip}$, *msg*) *packet* ⇒ *bool* **where**
*is-ftp-close* = (λ *i p*. (*id p* = *i* ∧ *content p* = *ftp-close*))

**definition**
*port-open* :: (*adr*$_{ip}$, *msg*) *history* ⇒ *id* ⇒ *port* ⇒ *bool* **where**
*port-open* = (λ *L a p*. (*not-before* (*is-ftp-close a*) (*is-ftp-port-request a p*) *L*))

**definition**
*is-ftp-other* :: *id* ⇒ (*adr*$_{ip}$, *msg* ) *packet* ⇒ *bool* **where**
*is-ftp-other* = (λ *i p*. (*id p* = *i* ∧ *content p* = *ftp-other*))

**fun** *are-ftp-other* **where**
*are-ftp-other i* (*x*#*xs*) = (*is-ftp-other i x* ∧ *are-ftp-other i xs*)
|*are-ftp-other i* [] = *True*

### 9.2.2. The protocol policy specification

We now have to model the respective state transitions. It is important to note that state transitions themselves allow all packets which are allowed by the policy, not only those which are allowed by the protocol. Their only task is to change the policy. As an alternative, we could have decided that they only allow packets which follow the protocol (e.g. come on the correct ports), but this should in our view rather be reflected in the policy itself.

Of course, not every message changes the policy. In such cases, we do not have to model different cases, one is enough. In our example, only messages 2 and 4 need special transitions. The default says that if the policy accepts the packet, it is added to the history, otherwise it is simply dropped. The policy remains the same in both cases.

**fun** *last-opened-port* **where**
*last-opened-port i* ((*j*,*s*,*d*,*ftp-port-request p*)#*xs*) = (*if i*=*j then p else last-opened-port i xs*)
| *last-opened-port i* (*x*#*xs*) = *last-opened-port i xs*
| *last-opened-port x* [] = *undefined*

**fun** *FTP-STA* :: ((*adr*$_{ip}$,*msg*) *history*, *adr*$_{ip}$, *msg*) *FWStateTransition*
**where**

*FTP-STA* ((*i*,*s*,*d*,*ftp-port-request pr*), (*log* , *pol*)) =
(*if before*(*Not o is-ftp-close i*)(*is-init i*) *log* ∧
*dest-port* (*i*,*s*,*d*,*ftp-port-request pr*) = (*21*::*port*)
*then Some* (((*i*,*s*,*d*,*ftp-port-request pr*)#*log* ,
(*allow-from-to-port pr* (*subnet-of d*) (*subnet-of s*)) ⊕ *pol*))
*else Some* (((*i*,*s*,*d*,*ftp-port-request pr*)#*log* ,*pol*)))

$|FTP\text{-}STA\ ((i,s,d,ftp\text{-}close),\ (log\ ,pol)) =$
$\quad (if\quad (\exists\ p.\ port\text{-}open\ log\ i\ p) \wedge dest\text{-}port\ (i,s,d,ftp\text{-}close) = (21::port)$
$\quad\ then\ Some\ ((i,s,d,ftp\text{-}close)\#log\ ,$
$\qquad\qquad deny\text{-}from\text{-}to\text{-}port\ (last\text{-}opened\text{-}port\ i\ log_)\ (subnet\text{-}of\ d)(subnet\text{-}of\ s) \bigoplus pol)$
$\quad else\ \ Some\ (((i,s,d,ftp\text{-}close)\#log\ ,\ pol)))$

$|FTP\text{-}STA\ (p,\ s) = Some\ (p\#(fst\ s),snd\ s)$

**fun** *FTP-STD* ::
$((adr_{ip},msg)\ history,\ adr_{ip},\ msg)\ FWStateTransition$
**where**
$FTP\text{-}STD\ (p,s) = Some\ s$

**definition** *TRPolicy* :: $(adr_{ip},msg)packet \times (adr_{ip},msg)history \times ((adr_{ip},msg)packet \mapsto unit)$
$\qquad\qquad \mapsto (unit \qquad\qquad \times (adr_{ip},msg)history \times ((adr_{ip},msg)packet \mapsto unit))$
**where** $TRPolicy = ((FTP\text{-}STA,FTP\text{-}STD) \bigotimes_{\nabla}\ applyPolicy)\ o\ (\lambda(x,(y,z)).((x,z),(x,(y,z))))$

**definition** $TRPolicy_{Mon}$
**where** $TRPolicy_{Mon} = policy2MON(TRPolicy)$

If required to contain the policy in the output

**definition** ${TRPolicy_{Mon}}'$
**where** ${TRPolicy_{Mon}}' = policy2MON\ (((\lambda(x,y,z).\ (z,(y,z)))\ o\text{-}f\ \ TRPolicy\ ))$

Now we specify our test scenario in more detail. We could test:

- one correct FTP-Protocol run,
- several runs after another,
- several runs interleaved,
- an illegal protocol run, or
- several illegal protocol runs.

We only do the the simplest case here: one correct protocol run.

There are four different states which are modelled as a datatype.

**datatype** *ftp-states* = *S0* | *S1* | *S2* | *S3*

The following constant is *True* for all sets which are correct FTP runs for a given source and destination address, ID, and data-port number.

**fun**
$is\text{-}ftp :: ftp\text{-}states \Rightarrow adr_{ip} \Rightarrow adr_{ip} \Rightarrow id \Rightarrow port \Rightarrow$
$\qquad (adr_{ip},msg)\ history \Rightarrow bool$

**where**
*is-ftp H c s i p [] = (H=S3)*
|*is-ftp H c s i p (x#InL) = (snd s = 21 ∧((λ (id,sr,de,co). (((id = i ∧ (*
*(H=ftp-states.S2 ∧ sr = c ∧ de = s ∧ co = ftp-init ∧ is-ftp S3 c s i p InL) ∨*
*(H=ftp-states.S1 ∧ sr = c ∧ de = s ∧ co = ftp-port-request p ∧ is-ftp S2 c s i p InL) ∨*
*(H=ftp-states.S1 ∧ sr = s ∧ de = (fst c,p) ∧ co= ftp-data ∧ is-ftp S1 c s i p InL) ∨*
*(H=ftp-states.S0 ∧ sr = c ∧ de = s ∧ co = ftp-close ∧ is-ftp S1 c s i p InL) ))))) x))*

**definition** *is-single-ftp-run :: $adr_{ip}$ src ⇒ $adr_{ip}$ dest ⇒ id ⇒ port ⇒ ($adr_{ip}$,msg) history set*

**where** *is-single-ftp-run s d i p = {x. (is-ftp S0 s d i p x)}*

The following constant then returns a set of all the historys which denote such a normal behaviour FTP run, again for a given source and destination address, ID, and data-port.

The following definition returns the set of all possible interleaving of two correct FTP protocol runs.

**definition**
*ftp-2-interleaved :: $adr_{ip}$ src ⇒ $adr_{ip}$ dest ⇒ id ⇒ port ⇒*
*$adr_{ip}$ src ⇒ $adr_{ip}$ dest ⇒ id ⇒ port ⇒*
*($adr_{ip}$,msg) history set* **where**
*ftp-2-interleaved s1 d1 i1 p1 s2 d2 i2 p2 =*
*{x. (is-ftp S0 s1 d1 i1 p1 (packet-with-id x i1)) ∧*
*(is-ftp S0 s2 d2 i2 p2 (packet-with-id x i2))}*

**lemma** *subnetOf-lemma*: (a::*int*) ≠ (c::*int*) ⟹ ∀ x∈*subnet-of* (a, b::*port*). (c, d) ∉ x*
⟨*proof*⟩

**lemma** *subnetOf-lemma2*: ∀ x∈*subnet-of* (a::*int*, b::*port*). (a, b) ∈ x*
⟨*proof*⟩

**lemma** *subnetOf-lemma3*: (∃ x. x ∈ *subnet-of* (a::*int*, b::*port*))*
⟨*proof*⟩

**lemma** *subnetOf-lemma4*: ∃ x∈*subnet-of* (a::*int*, b::*port*). (a, c::*port*) ∈ x*
⟨*proof*⟩

**lemma** *port-open-lemma*: ¬ (*Ex* (*port-open* [] (x::*port*)))*
⟨*proof*⟩

**lemmas** *FTPLemmas = TRPolicy-def applyPolicy-def policy2MON-def*
*Let-def in-subnet-def src-def*
*dest-def subnet-of-def*
*is-init-def p-accept-def port-open-def is-ftp-data-def is-ftp-close-def*

*is-ftp-port-request-def*
*content-def PortCombinators*
*exI subnetOf-lemma*
*subnetOf-lemma2 subnetOf-lemma3 subnetOf-lemma4*
*NetworkCore.id-def* $adr_{ip}$*Lemmas port-open-lemma*
*bind-SE-def unit-SE-def valid-SE-def*

**end**

**theory** *FTP-WithPolicy*
**imports**
*FTP*
**begin**

FTP where the policy is part of the output.

**definition** $POL :: 'a \Rightarrow 'a$
**where** $POL\ x = x$

Variant 2 takes the policy into the output

**fun** *FTP-STP* ::
$((id \rightharpoonup port),\ adr_{ip},\ msg)$ *FWStateTransitionP*
**where**

*FTP-STP* $(i,s,d,$*ftp-port-request pr*$)$ $(ports,\ policy) =$
(*if p-accept* $(i,s,d,$*ftp-port-request pr*$)$ *policy then*
*Some* (*allow* (*POL* ((*allow-from-to-port pr* (*subnet-of d*) (*subnet-of s*)) $\bigoplus$ *policy*)),
( $(ports(i \mapsto pr)),$(*allow-from-to-port pr* (*subnet-of d*) (*subnet-of s*))
$\bigoplus$ *policy*))
*else* (*Some* (*deny* (*POL policy*),(*ports*,*policy*))))

|*FTP-STP* $(i,s,d,$*ftp-close*$)$ $(ports,policy) =$
(*if* (*p-accept* $(i,s,d,$*ftp-close*$)$ *policy*) *then*
*case ports i of*
*Some pr* $\Rightarrow$
*Some*(*allow* (*POL* (*deny-from-to-port pr* (*subnet-of d*) (*subnet-of s*) $\bigoplus$ *policy*)),
*ports*($i$:=*None*),
*deny-from-to-port pr* (*subnet-of d*) (*subnet-of s*) $\bigoplus$ *policy*)
|*None* $\Rightarrow$*Some*(*allow* (*POL policy*), *ports*, *policy*)
*else Some* (*deny* (*POL policy*), *ports*, *policy*))

```
|FTP-STP p x =  (if p-accept p (snd x)
                then Some (allow (POL (snd x)),((fst x),snd x))
                else Some (deny (POL (snd x)),(fst x,snd x)))
```

**end**

## 9.3. VoIP Protocol

**theory** *VOIP*
**imports** *Stateful*
**begin**

After the FTP-Protocol which was rather simple we show the strength of the model with a more current and especially much more complicated example, namely Voice over IP (VoIP). VoIP is standardized by the ITU-T under the name H.323, which can be seen as an "umbrella standard" which aggregates standards for multimedia conferencing over packet-based networks (for a good overview of the protocol suite, see [**?**]). H.323 poses many problems to firewalls. These problems include (taken from [**?**]):

- An H.323 call is made up of many different simultaneous connections.
- Most connections are made to dynamic ports.
- The addresses and port numbers are exchanged within the data stream of the next higher connection.
- Calls can be initiated from outside the firewall.

Again we only consider a simplified VoIP scenario with the following seven messages which are grouped into four subprotocols (see Figure **??**):

- Registration and Admission (H.225, port 1719): The caller contacts its gatekeeper with a call request. The gatekeeper either rejects or confirms the request, returning the address of the callee in the latter case.
  - Admission Request (ARQ)
  - Admission Reject (ARJ)
  - Admission Confirm (ACF) $'a$
- Call Signaling (Q.931, port 1720) The caller and the callee agree on the dynamic ports over which the call will take place.
  - Setup *port*
  - Connect *port*

- Stream (dynamic ports). The call itself. In reality, several connections are used here.
- Fin (port 1720).

The two main differences to FTP are:

- In VoIP, we deal with three different entities: the caller, the callee, and the gatekeeper.
- We do not know in advance which entity will close the connection.

We model the protocol as seen from a firewall at the caller, namely we are not interested in the messages from the callee to its gatekeeper. Incoming calls are not modelled either, they would require a different set of state transitions.

The content of a packet now consists of one of the seven messages or a default one. It is parameterized with the type of the address that the gatekeeper returns.

**datatype** *'a voip-msg* = *ARQ*
| *ACF 'a*
| *ARJ*
| *Setup port*
| *Connect port*
| *Stream*
| *Fin*
| *other*

As before, we need operators which check if a packet contains a specific content and ID, respectively if such a packet has appeared in the trace.

**definition**
*is-arq* :: *NetworkCore.id* ⇒ (*'a*::*adr*, *'b voip-msg*) *packet* ⇒ *bool* **where**
*is-arq i p* = (*NetworkCore.id p* = *i* ∧ *content p* = *ARQ*)

**definition**
*is-fin* :: *id* ⇒ (*'a*::*adr*, *'b voip-msg*) *packet* ⇒ *bool* **where**
*is-fin i p* = (*id p* = *i* ∧ *content p* = *Fin*)

**definition**
*is-connect* :: *id* ⇒ *port* ⇒ (*'a*::*adr*, *'b voip-msg*) *packet* ⇒ *bool* **where**
*is-connect i port p* = (*id p* = *i* ∧ *content p* = *Connect port*)

**definition**
*is-setup* :: *id* ⇒ *port* ⇒ (*'a*::*adr*, *'b voip-msg*) *packet* ⇒ *bool* **where**
*is-setup i port p* = (*id p* = *i* ∧ *content p* = *Setup port*)

We need also an operator *ports-open* to get access to the two dynamic ports.

**definition**
*ports-open* :: *id* ⇒ *port* × *port* ⇒ (*adr*$_{ip}$, *'a voip-msg*) *history* ⇒ *bool* **where**
*ports-open i p L* = ((*not-before* (*is-fin i*) (*is-setup i* (*fst p*)) *L*) ∧
*not-before* (*is-fin i*) (*is-connect i* (*snd p*)) *L*)

As we do not know which entity closes the connection, we define an operator which checks if the closer is the caller.

**fun**
*src-is-initiator* :: *id* ⇒ $adr_{ip}$ ⇒ ($adr_{ip}$,'*b voip-msg*) *history* ⇒ *bool* **where**
*src-is-initiator i a* [] = *False*
|*src-is-initiator i a* (*p*#*S*) = (((*id p* = *i*) ∧
(∃ *port*. *content p* = *Setup port*) ∧
((*fst* (*src p*) = *fst a*))) ∨
(*src-is-initiator i a S*))

The first state transition is for those messages which do not change the policy. In this scenario, this only happens for the Stream messages.

**definition** *subnet-of-adr* **where**
*subnet-of-adr x* = {{(*a*,*b*). *a* = *x*}}

**fun** *VOIP-STA* ::
(($adr_{ip}$,*address voip-msg*) *history*, $adr_{ip}$, *address voip-msg*) *FWStateTransition*
**where**

*VOIP-STA* ((*a*,*c*,*d*,*ARQ*), (*InL*, *policy*)) =
*Some* (((*a*,*c*,*d*, *ARQ*)#*InL*,
(*allow-from-to-port* (*1719*::*port*)(*subnet-of d*) (*subnet-of c*)) ⨁ *policy*))

|*VOIP-STA* ((*a*,*c*,*d*,*ARJ*), (*InL*, *policy*)) =
(*if* (*not-before* (*is-fin a*) (*is-arq a*) *InL*)
*then Some* (((*a*,*c*,*d*,*ARJ*)#*InL*,
*deny-from-to-port* (*14*::*port*) (*subnet-of c*) (*subnet-of d*) ⨁ *policy*))
*else Some* (((*a*,*c*,*d*,*ARJ*)#*InL*,*policy*)))

|*VOIP-STA* ((*a*,*c*,*d*,*ACF callee*), (*InL*, *policy*)) =
*Some* (((*a*,*c*,*d*,*ACF callee*)#*InL*,
*allow-from-to-port* (*1720*::*port*) (*subnet-of-adr callee*) (*subnet-of d*) ⨁
*allow-from-to-port* (*1720*::*port*) (*subnet-of d*) (*subnet-of-adr callee*) ⨁
*deny-from-to-port* (*1719*::*port*) (*subnet-of d*) (*subnet-of c*) ⨁
*policy*))

|*VOIP-STA* ((*a*,*c*,*d*, *Setup port*), (*InL*, *policy*)) =
*Some* (((*a*,*c*,*d*,*Setup port*)#*InL*,
*allow-from-to-port port* (*subnet-of d*) (*subnet-of c*) ⨁ *policy*))

|*VOIP-STA* ((*a*,*c*,*d*, *Connect port*), (*InL*, *policy*)) =
*Some* (((*a*,*c*,*d*,*Connect port*)#*InL*,
*allow-from-to-port port* (*subnet-of d*) (*subnet-of c*) ⨁ *policy*))

```
|VOIP-STA ((a,c,d,Fin), (InL,policy)) =
      (if ∃ p1 p2. ports-open a (p1,p2) InL then (
         (if src-is-initiator a c InL
         then (Some (((a,c,d,Fin)#InL,
(deny-from-to-port (1720::int) (subnet-of c) (subnet-of d) ) ⊕
(deny-from-to-port (snd (SOME p. ports-open a p InL))
                  (subnet-of c) (subnet-of d)) ⊕
(deny-from-to-port (fst (SOME p. ports-open a p InL))
                   (subnet-of d) (subnet-of c)) ⊕ policy)))

         else (Some (((a,c,d,Fin)#InL,
(deny-from-to-port (1720::int) (subnet-of c) (subnet-of d) ) ⊕
(deny-from-to-port (fst (SOME p. ports-open a p InL))
                  (subnet-of c) (subnet-of d)) ⊕
(deny-from-to-port (snd (SOME p. ports-open a p InL))
                   (subnet-of d) (subnet-of c)) ⊕ policy)))))

      else
         (Some (((a,c,d,Fin)#InL,policy))))



| VOIP-STA (p, (InL, policy)) =
                     Some ((p#InL,policy))

fun VOIP-STD where
 VOIP-STD (p,s) = Some s


definition VOIP-TRPolicy where
 VOIP-TRPolicy = policy2MON (
  ((VOIP-STA,VOIP-STD) ⊗▽ applyPolicy) o (λ (x,(y,z)). ((x,z),(x,(y,z)))))
```

For a full protocol run, six states are needed.

```
datatype voip-states = S0 | S1 | S2 | S3 | S4 | S5
```

The constant *is-voip* checks if a trace corresponds to a legal VoIP protocol, given the IP-addresses of the three entities, the ID, and the two dynamic ports.

```
fun is-voip :: voip-states ⇒ address ⇒ address ⇒ address ⇒ id ⇒ port ⇒
               port ⇒ (adr_ip, address voip-msg) history ⇒ bool
where
 is-voip H s d g i p1 p2 [] = (H = S5)
|is-voip H s d g i p1 p2 (x#InL) =
  (((λ (id,sr,de,co).
 (((id = i ∧
(H = S4 ∧ ((sr = (s,1719) ∧ de = (g,1719) ∧ co = ARQ ∧
    is-voip S5 s d g i p1 p2 InL))) ∨
(H = S0 ∧ sr = (g,1719) ∧ de = (s,1719) ∧ co = ARJ ∧
```

*is-voip S4 s d g i p1 p2 InL*) ∨
(*H* = *S3* ∧ *sr* = (*g*,*1719*) ∧ *de* = (*s*,*1719*) ∧ *co* = *ACF d* ∧
*is-voip S4 s d g i p1 p2 InL*) ∨
(*H* = *S2* ∧ *sr* = (*s*,*1720*) ∧ *de* = (*d*,*1720*) ∧ *co* = *Setup p1* ∧
*is-voip S3 s d g i p1 p2 InL*) ∨
(*H* = *S1* ∧ *sr* = (*d*,*1720*) ∧ *de* = (*s*,*1720*) ∧ *co* = *Connect p2* ∧
*is-voip S2 s d g i p1 p2 InL*) ∨
(*H* = *S1* ∧ *sr* = (*s*,*p1*) ∧ *de* = (*d*,*p2*) ∧ *co* = *Stream* ∧
*is-voip S1 s d g i p1 p2 InL*) ∨
(*H* = *S1* ∧ *sr* = (*d*,*p2*) ∧ *de* = (*s*,*p1*) ∧ *co* = *Stream* ∧
*is-voip S1 s d g i p1 p2 InL*) ∨
(*H* = *S0* ∧ *sr* = (*d*,*1720*) ∧ *de* = (*s*,*1720*) ∧ *co* = *Fin* ∧
*is-voip S1 s d g i p1 p2 InL*) ∨
(*H* = *S0* ∧ *sr* = (*s*,*1720*) ∧ *de* = (*d*,*1720*) ∧ *co* = *Fin* ∧
*is-voip S1 s d g i p1 p2 InL*)))))) *x*)

Finally, *NB-voip* returns the set of protocol traces which correspond to a correct protocol run given the three addresses, the ID, and the two dynamic ports.

**definition**
*NB-voip* :: *address* ⇒ *address* ⇒ *address* ⇒ *id* ⇒ *port* ⇒ *port* ⇒
(*adr*$_{ip}$, *address voip-msg*) *history set* **where**
*NB-voip s d g i p1 p2*= {*x*. (*is-voip S0 s d g i p1 p2 x*)}

**end**

## 9.4. FTP and VoIP Protocol Interleaved

**theory** *FTPVOIP*
**imports**
*FTP-WithPolicy VOIP*
**begin**

**datatype** *ftpvoip* = *ARQ*
| *ACF int*
| *ARJ*
| *Setup port*
| *Connect port*
| *Stream*
| *Fin*
| *ftp-init*
| *ftp-port-request port*
| *ftp-data*
| *ftp-close*
| *other*

We now also make use of the ID field of a packet. It is used as session ID and we make

the assumption that they are all unique among different protocol runs.

At first, we need some predicates which check if a packet is a specific FTP message and has the correct session ID.

**definition**
*FTPVOIP-is-init* :: *id* ⇒ (*adr*$_{ip}$, *ftpvoip* ) *packet* ⇒ *bool* **where**
*FTPVOIP-is-init* = (λ *i p*. (*id p* = *i* ∧ *content p* = *ftp-init*))

**definition**
*FTPVOIP-is-port-request* :: *id* ⇒ *port* ⇒(*adr*$_{ip}$, *ftpvoip*) *packet* ⇒ *bool* **where**
*FTPVOIP-is-port-request* = (λ *i port p*. (*id p* = *i* ∧ *content p* = *ftp-port-request port*))

**definition**
*FTPVOIP-is-data* :: *id* ⇒ (*adr*$_{ip}$, *ftpvoip*) *packet* ⇒ *bool* **where**
*FTPVOIP-is-data* = (λ *i p*. (*id p* = *i* ∧ *content p* = *ftp-data*))

**definition**
*FTPVOIP-is-close* :: *id* ⇒ (*adr*$_{ip}$, *ftpvoip*) *packet* ⇒ *bool* **where**
*FTPVOIP-is-close* = (λ *i p*. (*id p* = *i* ∧ *content p* = *ftp-close*))

**definition**
*FTPVOIP-port-open* :: (*adr*$_{ip}$, *ftpvoip*) *history* ⇒ *id* ⇒ *port* ⇒ *bool* **where**
*FTPVOIP-port-open* = (λ *L a p*. (*not-before* (*FTPVOIP-is-close a*) (*FTPVOIP-is-port-request a p*) *L*))

**definition**
*FTPVOIP-is-other* :: *id* ⇒ (*adr*$_{ip}$, *ftpvoip* ) *packet* ⇒ *bool* **where**
*FTPVOIP-is-other* = (λ *i p*. (*id p* = *i* ∧ *content p* = *other*))

**fun** *FTPVOIP-are-other* **where**
*FTPVOIP-are-other i* (*x*#*xs*) = (*FTPVOIP-is-other i x* ∧ *FTPVOIP-are-other i xs*)
|*FTPVOIP-are-other i* [] = *True*

**fun** *last-opened-port* **where**
*last-opened-port i* ((*j*,*s*,*d*,*ftp-port-request p*)#*xs*) = (*if i*=*j then p else last-opened-port i xs*)
| *last-opened-port i* (*x*#*xs*) = *last-opened-port i xs*
| *last-opened-port x* [] = *undefined*

**fun** *FTPVOIP-FTP-STA* ::
((*adr*$_{ip}$, *ftpvoip*) *history*, *adr*$_{ip}$, *ftpvoip*) *FWStateTransition*
**where**

*FTPVOIP-FTP-STA* ((*i*,*s*,*d*,*ftp-port-request pr*), (*InL*, *policy*)) =
(*if not-before* (*FTPVOIP-is-close i*) (*FTPVOIP-is-init i*) *InL* ∧
*dest-port* (*i*,*s*,*d*,*ftp-port-request pr*) = (*21*::*port*) *then*
*Some* (((*i*,*s*,*d*,*ftp-port-request pr*)#*InL*, *policy* ++

```
                    (allow-from-to-port pr (subnet-of d) (subnet-of s))))
    else Some (((i,s,d,ftp-port-request pr)#InL,policy)))

 |FTPVOIP-FTP-STA ((i,s,d,ftp-close), (InL,policy)) =
    (if  (∃ p. FTPVOIP-port-open InL i p) ∧ dest-port (i,s,d,ftp-close) = (21::port)
     then Some ((i,s,d,ftp-close)#InL, policy ++
          deny-from-to-port (last-opened-port i InL) (subnet-of d) (subnet-of s))
    else  Some (((i,s,d,ftp-close)#InL, policy)))


|FTPVOIP-FTP-STA (p, s) = Some (p#(fst s),snd s)


fun FTPVOIP-FTP-STD ::
  ((adr_ip, ftpvoip) history, adr_ip,  ftpvoip) FWStateTransition
where
 FTPVOIP-FTP-STD (p,s) = Some s
```

**definition**
*FTPVOIP-is-arq* :: *NetworkCore.id* ⇒ ('*a*::*adr*, *ftpvoip*) *packet* ⇒ *bool* **where**
*FTPVOIP-is-arq i p* = (*NetworkCore.id p* = *i* ∧ *content p* = *ARQ*)

**definition**
*FTPVOIP-is-fin* :: *id* ⇒ ('*a*::*adr*, *ftpvoip*) *packet* ⇒ *bool* **where**
*FTPVOIP-is-fin i p* = (*id p* = *i* ∧ *content p* = *Fin*)

**definition**
*FTPVOIP-is-connect* :: *id* ⇒ *port* ⇒ ('*a*::*adr*, *ftpvoip*) *packet* ⇒ *bool* **where**
*FTPVOIP-is-connect i port p* = (*id p* = *i* ∧ *content p* = *Connect port*)

**definition**
*FTPVOIP-is-setup* :: *id* ⇒ *port* ⇒ ('*a*::*adr*, *ftpvoip*) *packet* ⇒ *bool* **where**
*FTPVOIP-is-setup i port p* = (*id p* = *i* ∧ *content p* = *Setup port*)

We need also an operator *ports-open* to get access to the two dynamic ports.

**definition**
*FTPVOIP-ports-open* :: *id* ⇒ *port* × *port* ⇒ (*adr*$_{ip}$, *ftpvoip*) *history* ⇒ *bool* **where**
*FTPVOIP-ports-open i p L* = ((*not-before* (*FTPVOIP-is-fin i*) (*FTPVOIP-is-setup i* (*fst p*)) *L*) ∧
*not-before* (*FTPVOIP-is-fin i*) (*FTPVOIP-is-connect i* (*snd p*)) *L*)

As we do not know which entity closes the connection, we define an operator which checks if the closer is the caller.

**fun**
*FTPVOIP-src-is-initiator* :: *id* ⇒ *adr*$_{ip}$ ⇒ (*adr*$_{ip}$,*ftpvoip*) *history* ⇒ *bool* **where**
*FTPVOIP-src-is-initiator i a* [] = *False*
|*FTPVOIP-src-is-initiator i a* (*p*#*S*) = (((*id p* = *i*) ∧

$(\exists\ port.\ content\ p = Setup\ port) \wedge$
$((fst\ (src\ p) = fst\ a))) \vee$
$(FTPVOIP\text{-}src\text{-}is\text{-}initiator\ i\ a\ S))$

**definition** *FTPVOIP-subnet-of-adr* :: $int \Rightarrow adr_{ip}\ net$ **where**
*FTPVOIP-subnet-of-adr* $x = \{\{(a,b).\ a = x\}\}$

**fun** *FTPVOIP-VOIP-STA* ::
$((adr_{ip}, ftpvoip)\ history,\ adr_{ip},\ ftpvoip)\ FWStateTransition$
**where**

*FTPVOIP-VOIP-STA* $((a,c,d,ARQ), (InL, policy)) =$
*Some* $(((a,c,d, ARQ)\#InL,$
$(allow\text{-}from\text{-}to\text{-}port\ (1719::port)(subnet\text{-}of\ d)\ (subnet\text{-}of\ c)) \bigoplus\ policy))$

$|$*FTPVOIP-VOIP-STA* $((a,c,d,ARJ), (InL, policy)) =$
$(if\ (not\text{-}before\ (FTPVOIP\text{-}is\text{-}fin\ a)\ (FTPVOIP\text{-}is\text{-}arq\ a)\ InL)$
$then\ Some\ (((a,c,d,ARJ)\#InL,$
$deny\text{-}from\text{-}to\text{-}port\ (14::port)\ (subnet\text{-}of\ c)\ (subnet\text{-}of\ d) \bigoplus\ policy))$
$else\ Some\ (((a,c,d,ARJ)\#InL,policy)))$

$|$*FTPVOIP-VOIP-STA* $((a,c,d,ACF\ callee), (InL, policy)) =$
$Some\ (((a,c,d,ACF\ callee)\#InL,$
$allow\text{-}from\text{-}to\text{-}port\ (1720::port)\ (subnet\text{-}of\text{-}adr\ callee)\ (subnet\text{-}of\ d) \bigoplus$
$allow\text{-}from\text{-}to\text{-}port\ (1720::port)\ (subnet\text{-}of\ d)\ (subnet\text{-}of\text{-}adr\ callee) \bigoplus$
$deny\text{-}from\text{-}to\text{-}port\ (1719::port)\ (subnet\text{-}of\ d)\ (subnet\text{-}of\ c) \bigoplus$
$policy))$

$|$*FTPVOIP-VOIP-STA* $((a,c,d,\ Setup\ port), (InL, policy)) =$
$Some\ (((a,c,d,Setup\ port)\#InL,$
$allow\text{-}from\text{-}to\text{-}port\ port\ (subnet\text{-}of\ d)\ (subnet\text{-}of\ c) \bigoplus\ policy))$

$|$*FTPVOIP-VOIP-STA* $((a,c,d,\ ftpvoip.Connect\ port), (InL, policy)) =$
$Some\ (((a,c,d,ftpvoip.Connect\ port)\#InL,$
$allow\text{-}from\text{-}to\text{-}port\ port\ (subnet\text{-}of\ d)\ (subnet\text{-}of\ c)\ \bigoplus\ policy))$

$|$*FTPVOIP-VOIP-STA* $((a,c,d,Fin), (InL,policy)) =$
$(if\ \exists\ p1\ p2.\ FTPVOIP\text{-}ports\text{-}open\ a\ (p1,p2)\ InL\ then\ ($
$(if\ FTPVOIP\text{-}src\text{-}is\text{-}initiator\ a\ c\ InL$
$then\ (Some\ (((a,c,d,Fin)\#InL,$
$(deny\text{-}from\text{-}to\text{-}port\ (1720::int)\ (subnet\text{-}of\ c)\ (subnet\text{-}of\ d)\ ) \bigoplus$
$(deny\text{-}from\text{-}to\text{-}port\ (snd\ (SOME\ p.\ FTPVOIP\text{-}ports\text{-}open\ a\ p\ InL))$
$(subnet\text{-}of\ c)\ (subnet\text{-}of\ d)) \bigoplus$

```
(deny-from-to-port (fst (SOME p. FTPVOIP-ports-open a p InL))
                   (subnet-of d) (subnet-of c)) ⊕ policy)))

          else (Some (((a,c,d,Fin)#InL,
(deny-from-to-port (1720::int) (subnet-of c) (subnet-of d) ) ⊕
(deny-from-to-port (fst (SOME p. FTPVOIP-ports-open a p InL))
                   (subnet-of c) (subnet-of d)) ⊕
(deny-from-to-port (snd (SOME p. FTPVOIP-ports-open a p InL))
                   (subnet-of d) (subnet-of c)) ⊕ policy)))))

       else
          (Some (((a,c,d,Fin)#InL,policy))))



| FTPVOIP-VOIP-STA (p, (InL, policy)) =
                        Some ((p#InL,policy))



fun FTPVOIP-VOIP-STD ::
  ((adr_ip, ftpvoip) history, adr_ip, ftpvoip) FWStateTransition
where
 FTPVOIP-VOIP-STD (p,s) = Some s




definition FTP-VOIP-STA :: ((adr_ip, ftpvoip) history, adr_ip, ftpvoip) FWStateTransition
where
 FTP-VOIP-STA = (λ(x,x). Some x) o-m ((FTPVOIP-FTP-STA ⊗_S FTPVOIP-VOIP-STA
o (λ (p,x). (p,x,x))))


definition FTP-VOIP-STD :: ((adr_ip, ftpvoip) history, adr_ip, ftpvoip) FWStateTransition
where
 FTP-VOIP-STD = (λ(x,x). Some x) o-m ((FTPVOIP-FTP-STD ⊗_S FTPVOIP-VOIP-STD
o (λ (p,x). (p,x,x))))



definition FTPVOIP-TRPolicy where
 FTPVOIP-TRPolicy = policy2MON (
   (((FTP-VOIP-STA,FTP-VOIP-STD) ⊗_▽ applyPolicy) o (λ (x,(y,z)). ((x,z),(x,(y,z))))))
```

**lemmas** *FTPVOIP-ST-simps = Let-def in-subnet-def src-def dest-def*
*subnet-of-def id-def FTPVOIP-port-open-def*
*FTPVOIP-is-init-def FTPVOIP-is-data-def FTPVOIP-is-port-request-def FTPVOIP-is-close-def*
*p-accept-def content-def PortCombinators exI*
*NetworkCore.id-def* $adr_{ip}$*Lemmas*

**datatype** *ftp-states2 = FS0 | FS1 | FS2 | FS3*

**datatype** *voip-states2 = V0 | V1 | V2 | V3 | V4 | V5*

The constant *is-voip* checks if a trace corresponds to a legal VoIP protocol, given the IP-addresses of the three entities, the ID, and the two dynamic ports.

**fun** *FTPVOIP-is-voip* :: *voip-states2* ⇒ *address* ⇒ *address* ⇒ *address* ⇒ *id* ⇒ *port* ⇒
*port* ⇒ ($adr_{ip}$, *ftpvoip*) *history* ⇒ *bool*
**where**
*FTPVOIP-is-voip H s d g i p1 p2* [] = (*H = V5*)
|*FTPVOIP-is-voip H s d g i p1 p2* (*x*#*InL*) =
(((λ (*id*,*sr*,*de*,*co*).
(((*id = i* ∧
(*H = V4* ∧ ((*sr* = (*s*,*1719*) ∧ *de* = (*g*,*1719*) ∧ *co = ARQ* ∧
*FTPVOIP-is-voip V5 s d g i p1 p2 InL*))) ∨
(*H = V0* ∧ *sr* = (*g*,*1719*) ∧ *de* = (*s*,*1719*) ∧ *co = ARJ* ∧
*FTPVOIP-is-voip V4 s d g i p1 p2 InL*) ∨
(*H = V3* ∧ *sr* = (*g*,*1719*) ∧ *de* = (*s*,*1719*) ∧ *co = ACF d* ∧
*FTPVOIP-is-voip V4 s d g i p1 p2 InL*) ∨
(*H = V2* ∧ *sr* = (*s*,*1720*) ∧ *de* = (*d*,*1720*) ∧ *co = Setup p1* ∧
*FTPVOIP-is-voip V3 s d g i p1 p2 InL*) ∨
(*H = V1* ∧ *sr* = (*d*,*1720*) ∧ *de* = (*s*,*1720*) ∧ *co = Connect p2* ∧
*FTPVOIP-is-voip V2 s d g i p1 p2 InL*) ∨
(*H = V1* ∧ *sr* = (*s*,*p1*) ∧ *de* = (*d*,*p2*) ∧ *co = Stream* ∧
*FTPVOIP-is-voip V1 s d g i p1 p2 InL*) ∨
(*H = V1* ∧ *sr* = (*d*,*p2*) ∧ *de* = (*s*,*p1*) ∧ *co = Stream* ∧
*FTPVOIP-is-voip V1 s d g i p1 p2 InL*) ∨
(*H = V0* ∧ *sr* = (*d*,*1720*) ∧ *de* = (*s*,*1720*) ∧ *co = Fin* ∧
*FTPVOIP-is-voip V1 s d g i p1 p2 InL*) ∨
(*H = V0* ∧ *sr* = (*s*,*1720*) ∧ *de* = (*d*,*1720*) ∧ *co = Fin* ∧
*FTPVOIP-is-voip V1 s d g i p1 p2 InL*))))) *x*)

Finally, *NB-voip* returns the set of protocol traces which correspond to a correct protocol run given the three addresses, the ID, and the two dynamic ports.

**definition**
*FTPVOIP-NB-voip* :: *address* ⇒ *address* ⇒ *address* ⇒ *id* ⇒ *port* ⇒ *port* ⇒
($adr_{ip}$, *ftpvoip*) *history set* **where**
*FTPVOIP-NB-voip s d g i p1 p2*= {*x*. (*FTPVOIP-is-voip V0 s d g i p1 p2 x*)}

**fun**
*FTPVOIP-is-ftp* :: *ftp-states2* ⇒ $adr_{ip}$ ⇒ $adr_{ip}$ ⇒ *id* ⇒ *port* ⇒
($adr_{ip}$, *ftpvoip*) *history* ⇒ *bool*
**where**
*FTPVOIP-is-ftp H c s i p* [] = (*H*=*FS3*)
|*FTPVOIP-is-ftp H c s i p* (*x*#*InL*) = (*snd s* = *21* ∧((λ (*id*,*sr*,*de*,*co*). (((*id* = *i* ∧ (
(*H*=*FS2* ∧ *sr* = *c* ∧ *de* = *s* ∧ *co* = *ftp-init* ∧ *FTPVOIP-is-ftp FS3 c s i p InL*) ∨
(*H*=*FS1* ∧ *sr* = *c* ∧ *de* = *s* ∧ *co* = *ftp-port-request p* ∧ *FTPVOIP-is-ftp FS2 c s i p InL*)
∨
(*H*=*FS1* ∧ *sr* = *s* ∧ *de* = (*fst c*,*p*) ∧ *co*= *ftp-data* ∧ *FTPVOIP-is-ftp FS1 c s i p InL*) ∨
(*H*=*FS0* ∧ *sr* = *c* ∧ *de* = *s* ∧ *co* = *ftp-close* ∧ *FTPVOIP-is-ftp FS1 c s i p InL*) ))))) *x*))

**definition**
*FTPVOIP-NB-ftp* :: $adr_{ip}$ *src* ⇒ $adr_{ip}$ *dest* ⇒ *id* ⇒ *port* ⇒
($adr_{ip}$, *ftpvoip*) *history set* **where**
*FTPVOIP-NB-ftp s d i p* = {*x*. (*FTPVOIP-is-ftp FS0 s d i p x*)}

**definition**
*ftp-voip-interleaved* :: $adr_{ip}$ *src* ⇒ $adr_{ip}$ *dest* ⇒ *id* ⇒ *port* ⇒
*address* ⇒ *address* ⇒ *address* ⇒ *id* ⇒ *port* ⇒ *port* ⇒
($adr_{ip}$, *ftpvoip*) *history set*

**where**
*ftp-voip-interleaved s1 d1 i1 p1 vs vd vg vi vp1 vp2* =
{*x*. (*FTPVOIP-is-ftp FS0 s1 d1 i1 p1* (*packet-with-id x i1*)) ∧
(*FTPVOIP-is-voip V0 vs vd vg vi vp1 vp2* (*packet-with-id x vi*))}

**end**

# A. Appendix

**theory** *NormalisationGenericProofs*
**imports** *FWNormalisationCore*
**begin**

This theory contains the generic proofs of the normalisation procedure, i.e. those which are independent from the concrete semantical interpretation function.

**lemma** *domNMT*: $dom\ X \neq \{\} \Longrightarrow X \neq \emptyset$
⟨*proof*⟩

**lemma** *denyNMT*: $deny\text{-}all \neq \emptyset$
⟨*proof*⟩

**lemma** *wellformed-policy1-charn*[*rule-format*] : $wellformed\text{-}policy1\ p \longrightarrow DenyAll \in set\ p \longrightarrow (\exists\ p'.\ p = DenyAll\ \#\ p' \wedge DenyAll \notin set\ p')$
⟨*proof*⟩

**lemma** *singleCombinatorsConc*: $singleCombinators\ (x\#xs) \Longrightarrow singleCombinators\ xs$
⟨*proof*⟩

**lemma** *aux0-0*: $singleCombinators\ x \Longrightarrow \neg\ (\exists\ a\ b.\ (a \oplus b) \in set\ x)$
⟨*proof*⟩

**lemma** *aux0-4*: $(a \in set\ x \vee a \in set\ y) = (a \in set\ (x@y))$
⟨*proof*⟩

**lemma** *aux0-1*: $[\![singleCombinators\ xs;\ singleCombinators\ [x]]\!] \Longrightarrow singleCombinators\ (x\#xs)$
⟨*proof*⟩

**lemma** *aux0-6*: $[\![singleCombinators\ xs;\ \neg\ (\exists\ a\ b.\ x = a \oplus b)]\!] \Longrightarrow singleCombinators(x\#xs)$
⟨*proof*⟩

**lemma** *aux0-5*: $\neg\ (\exists\ a\ b.\ (a \oplus b) \in set\ x) \Longrightarrow singleCombinators\ x$
⟨*proof*⟩

**lemma** *ANDConc*[*rule-format*]: $allNetsDistinct\ (a\#p) \longrightarrow allNetsDistinct\ (p)$
⟨*proof*⟩

**lemma** *aux6*: *twoNetsDistinct a1 a2 a b* $\Longrightarrow$
*dom* (*deny-all-from-to a1 a2*) $\cap$ *dom* (*deny-all-from-to a b*) = {}
⟨*proof*⟩

**lemma** *aux5*[*rule-format*]: (*DenyAllFromTo a b*) $\in$ *set p* $\longrightarrow$ *a* $\in$ *set* (*net-list p*)
⟨*proof*⟩

**lemma** *aux5a*[*rule-format*]: (*DenyAllFromTo b a*) $\in$ *set p* $\longrightarrow$ *a* $\in$ *set* (*net-list p*)
⟨*proof*⟩

**lemma** *aux5c*[*rule-format*]:
(*AllowPortFromTo a b po*) $\in$ *set p* $\longrightarrow$ *a* $\in$ *set* (*net-list p*)
⟨*proof*⟩

**lemma** *aux5d*[*rule-format*]:
(*AllowPortFromTo b a po*) $\in$ *set p* $\longrightarrow$ *a* $\in$ *set* (*net-list p*)
⟨*proof*⟩

**lemma** *aux10*[*rule-format*]: *a* $\in$ *set* (*net-list p*) $\longrightarrow$ *a* $\in$ *set* (*net-list-aux p*)
⟨*proof*⟩

**lemma** *srcInNetListaux*[*simp*]: ⟦*x* $\in$ *set p*; *singleCombinators*[*x*]; *x* $\neq$ *DenyAll*⟧ $\Longrightarrow$
*srcNet x* $\in$ *set* (*net-list-aux p*)
⟨*proof*⟩

**lemma** *destInNetListaux*[*simp*]: ⟦*x* $\in$ *set p*; *singleCombinators*[*x*]; *x* $\neq$ *DenyAll*⟧ $\Longrightarrow$
*destNet x* $\in$ *set* (*net-list-aux p*)
⟨*proof*⟩

**lemma** *tND1*: ⟦*allNetsDistinct p*; *x* $\in$ *set p*; *y* $\in$ *set p*; *a* = *srcNet x*;
*b* = *destNet x*; *c* = *srcNet y*; *d* = *destNet y*; *a* $\neq$ *c*;
*singleCombinators*[*x*]; *x* $\neq$ *DenyAll*; *singleCombinators*[*y*];
*y* $\neq$ *DenyAll*⟧ $\Longrightarrow$ *twoNetsDistinct a b c d*
⟨*proof*⟩

**lemma** *tND2*: ⟦*allNetsDistinct p*; *x* $\in$ *set p*; *y* $\in$ *set p*; *a* = *srcNet x*;
*b* = *destNet x*; *c* = *srcNet y*; *d* = *destNet y*; *b* $\neq$ *d*;
*singleCombinators*[*x*]; *x* $\neq$ *DenyAll*; *singleCombinators*[*y*];
*y* $\neq$ *DenyAll*⟧ $\Longrightarrow$ *twoNetsDistinct a b c d*
⟨*proof*⟩

**lemma** *tND*: ⟦*allNetsDistinct p*; *x* ∈ *set p*; *y* ∈ *set p*; *a* = *srcNet x*;
*b* = *destNet x*; *c* = *srcNet y*; *d* = *destNet y*; *a* ≠ *c* ∨ *b* ≠ *d*;
*singleCombinators*[*x*]; *x* ≠ *DenyAll*; *singleCombinators*[*y*]; *y* ≠ *DenyAll*⟧
⟹ *twoNetsDistinct a b c d*
⟨*proof*⟩

**lemma** *aux7*: ⟦*DenyAllFromTo a b* ∈ *set p*; *allNetsDistinct* ((*DenyAllFromTo c d*)#*p*);
*a*≠ *c*∨ *b*≠ *d*⟧ ⟹ *twoNetsDistinct a b c d*
⟨*proof*⟩

**lemma** *aux7a*: ⟦*DenyAllFromTo a b* ∈ *set p*;
*allNetsDistinct* ((*AllowPortFromTo c d po*)#*p*); *a* ≠ *c*∨ *b* ≠ *d*⟧ ⟹
*twoNetsDistinct a b c d*
⟨*proof*⟩

**lemma** *nDComm*: **assumes** *ab*: *netsDistinct a b* **shows** *ba*: *netsDistinct b a*
⟨*proof*⟩

**lemma** *tNDComm*:
**assumes** *abcd*: *twoNetsDistinct a b c d* **shows** *twoNetsDistinct c d a b*
⟨*proof*⟩

**lemma** *aux*[*rule-format*]: *a* ∈ *set* (*removeShadowRules2 p*) ⟶ *a* ∈ *set p*
⟨*proof*⟩

**lemma** *aux12*: ⟦*a* ∈ *x*; *b* ∉ *x*⟧ ⟹ *a* ≠ *b*
⟨*proof*⟩

**lemma** *ND0aux1*[*rule-format*]: *DenyAllFromTo x y* ∈ *set b* ⟹
*x* ∈ *set* (*net-list-aux b*)
⟨*proof*⟩

**lemma** *ND0aux2*[*rule-format*]: *DenyAllFromTo x y* ∈ *set b* ⟹
*y* ∈ *set* (*net-list-aux b*)
⟨*proof*⟩

**lemma** *ND0aux3*[*rule-format*]: *AllowPortFromTo x y p* ∈ *set b* ⟹
*x* ∈ *set* (*net-list-aux b*)
⟨*proof*⟩

**lemma** *ND0aux4*[*rule-format*]: *AllowPortFromTo x y p* ∈ *set b* ⟹
*y* ∈ *set* (*net-list-aux b*)
⟨*proof*⟩

**lemma** *aNDSubsetaux*[*rule-format*]: *singleCombinators a* $\longrightarrow$ *set a* $\subseteq$ *set b* $\longrightarrow$
*set* (*net-list-aux a*) $\subseteq$ *set* (*net-list-aux b*)
⟨*proof*⟩

**lemma** *aNDSetsEqaux*[*rule-format*]: *singleCombinators a* $\longrightarrow$ *singleCombinators b* $\longrightarrow$
*set a* = *set b* $\longrightarrow$ *set* (*net-list-aux a*) = *set* (*net-list-aux b*)
⟨*proof*⟩

**lemma** *aNDSubset*: ⟦*singleCombinators a*;*set a* $\subseteq$ *set b*; *allNetsDistinct b*⟧ $\Longrightarrow$
*allNetsDistinct a*
⟨*proof*⟩

**lemma** *aNDSetsEq*: ⟦*singleCombinators a*; *singleCombinators b*; *set a* = *set b*;
*allNetsDistinct b*⟧ $\Longrightarrow$ *allNetsDistinct a*
⟨*proof*⟩

**lemma** *SCConca*: ⟦*singleCombinators p*; *singleCombinators* [*a*]⟧ $\Longrightarrow$
*singleCombinators* (*a*#*p*)
⟨*proof*⟩

**lemma** *aux3*[*simp*]: ⟦*singleCombinators p*; *singleCombinators* [*a*];
*allNetsDistinct* (*a*#*p*)⟧ $\Longrightarrow$ *allNetsDistinct* (*a*#*a*#*p*)
⟨*proof*⟩

**lemma** *wp1-aux1a*[*rule-format*]: *xs* $\neq$ [] $\longrightarrow$ *wellformed-policy1-strong* (*xs* @ [*x*]) $\longrightarrow$
*wellformed-policy1-strong xs*
⟨*proof*⟩

**lemma** *wp1alternative-RS1*[*rule-format*]: *DenyAll* $\in$ *set p* $\longrightarrow$
*wellformed-policy1-strong* (*removeShadowRules1 p*)
⟨*proof*⟩

**lemma** *wellformed-eq*: *DenyAll* $\in$ *set p* $\longrightarrow$
((*wellformed-policy1 p*) = (*wellformed-policy1-strong p*))
⟨*proof*⟩

**lemma** *set-insort*: *set*(*insort x xs l*) = *insert x* (*set xs*)
⟨*proof*⟩

**lemma** *set-sort*[*simp*]: *set*(*sort xs l*) = *set xs*
⟨*proof*⟩

**lemma** *set-sortQ*: *set*(*qsort xs l*) = *set xs*
⟨*proof*⟩

**lemma** *aux79*[*rule-format*]: $y \in set\ (insort\ x\ a\ l) \longrightarrow\ y \neq x \longrightarrow y \in set\ a$
⟨*proof*⟩

**lemma** *aux80*: $[\![y \notin set\ p;\ y \neq x]\!] \Longrightarrow y \notin set\ (insort\ x\ (sort\ p\ l)\ l)$
⟨*proof*⟩

**lemma** *WP1Conca*: $DenyAll \notin set\ p \Longrightarrow wellformed\text{-}policy1\ (a\#p)$
⟨*proof*⟩

**lemma** *saux*[*simp*]: $(insort\ DenyAll\ p\ l) = DenyAll\#p$
⟨*proof*⟩

**lemma** *saux3*[*rule-format*]: $DenyAllFromTo\ a\ b \in set\ list \longrightarrow$
$DenyAllFromTo\ c\ d \notin set\ list \longrightarrow (a \neq c) \vee (b \neq d)$
⟨*proof*⟩

**lemma** *waux2*[*rule-format*]: $(DenyAll \notin set\ xs) \longrightarrow wellformed\text{-}policy1\ xs$
⟨*proof*⟩

**lemma** *waux3*[*rule-format*]: $[\![x \neq a;\ \ x \notin set\ p]\!] \Longrightarrow x \notin set\ (insort\ a\ p\ l)$
⟨*proof*⟩

**lemma** *wellformed1-sorted-aux*[*rule-format*]: $wellformed\text{-}policy1\ (x\#p) \Longrightarrow$
$wellformed\text{-}policy1\ (insort\ x\ p\ l)$
⟨*proof*⟩

**lemma** *wellformed1-sorted-auxQ*[*rule-format*]: $wellformed\text{-}policy1\ (p) \Longrightarrow$
$wellformed\text{-}policy1\ (qsort\ p\ l)$
⟨*proof*⟩

**lemma** *SR1Subset*: $set\ (removeShadowRules1\ p) \subseteq set\ p$
⟨*proof*⟩

**lemma** *SCSubset*[*rule-format*]: $singleCombinators\ b \longrightarrow set\ a \subseteq set\ b \longrightarrow$
$singleCombinators\ a$
⟨*proof*⟩

**lemma** *setInsert*[*simp*]: $set\ list \subseteq insert\ a\ (set\ list)$
⟨*proof*⟩

**lemma** *SC-RS1*[*rule-format*,*simp*]: $singleCombinators\ p \longrightarrow allNetsDistinct\ p \longrightarrow$
$singleCombinators\ (removeShadowRules1\ p)$
⟨*proof*⟩

**lemma** *RS2Set*[*rule-format*]: $set\ (removeShadowRules2\ p) \subseteq set\ p$

⟨*proof*⟩

**lemma** *WP1*: $a \notin set\ list \Longrightarrow a \notin set\ (removeShadowRules2\ list)$
⟨*proof*⟩

**lemma** *denyAllDom*[*simp*]: $x \in dom\ (deny\text{-}all)$
⟨*proof*⟩

**lemma** *lCdom2*: $(list2FWpolicy\ (a\ @\ (b\ @\ c))) = (list2FWpolicy\ ((a@b)@c))$
⟨*proof*⟩

**lemma** *SCConcEnd*: $singleCombinators\ (xs\ @\ [xa]) \Longrightarrow singleCombinators\ xs$
⟨*proof*⟩

**lemma** *list2FWpolicyconc*[*rule-format*]: $a \neq [] \longrightarrow$
$(list2FWpolicy\ (xa\ \#\ a)) = (xa) \oplus (list2FWpolicy\ a)$
⟨*proof*⟩

**lemma** *wp1n-tl* [*rule-format*]: $wellformed\text{-}policy1\text{-}strong\ p \longrightarrow$
$p = (DenyAll\#(tl\ p))$
⟨*proof*⟩

**lemma** *foo2*:
$[\![a \notin set\ ps;\ a \notin set\ ss;\ set\ p = set\ s;\ \ p = (a\#(ps));\ s = (a\#ss)]\!] \Longrightarrow$
$set\ (ps) = set\ (ss)$
⟨*proof*⟩

**lemma** *SCnotConc*[*rule-format*,*simp*]: $a \oplus b \in set\ p \longrightarrow singleCombinators\ p \longrightarrow False$
⟨*proof*⟩

**lemma** *auxx8*: $removeShadowRules1\text{-}alternative\text{-}rev\ [x] = [x]$
⟨*proof*⟩

**lemma** *RS1End*[*rule-format*]: $x \neq DenyAll \longrightarrow removeShadowRules1\ (xs\ @\ [x]) =$
$(removeShadowRules1\ xs)@[x]$
⟨*proof*⟩

**lemma** *aux114*: $x \neq DenyAll \Longrightarrow removeShadowRules1\text{-}alternative\text{-}rev\ (x\#xs) =$
$x\#(removeShadowRules1\text{-}alternative\text{-}rev\ xs)$
⟨*proof*⟩

**lemma** *aux115*[*rule-format*]: $x \neq DenyAll \Longrightarrow removeShadowRules1\text{-}alternative\ (xs@[x])$
$= (removeShadowRules1\text{-}alternative\ xs)@[x]$
⟨*proof*⟩

**lemma** *RS1-DA*[*simp*]: $removeShadowRules1\ (xs\ @\ [DenyAll]) = [DenyAll]$

⟨*proof*⟩

**lemma** *rSR1-eq*: *removeShadowRules1-alternative* = *removeShadowRules1*
⟨*proof*⟩

**lemma** *domInterMT*[*rule-format*]: ⟦*dom a* ∩ *dom b* = {}; *x* ∈ *dom a*⟧ ⟹ *x* ∉ *dom b*
⟨*proof*⟩

**lemma** *domComm*: *dom a* ∩ *dom b* = *dom b* ∩ *dom a*
⟨*proof*⟩

**lemma** *r-not-DA-in-tl*[*rule-format*]: *wellformed-policy1-strong p* ⟶ *a* ∈ *set p* ⟶
*a* ≠ *DenyAll* ⟶ *a* ∈ *set* (*tl p*)
⟨*proof*⟩

**lemma** *wp1-aux1aa*[*rule-format*]: *wellformed-policy1-strong p* ⟶ *DenyAll* ∈ *set p*
⟨*proof*⟩

**lemma** *mauxa*: (∃ *r*. *a b* = ⌊*r*⌋) = (*a b* ≠ ⊥)
⟨*proof*⟩

**lemma** *l2p-aux*[*rule-format*]: *list* ≠ [] ⟶
*list2FWpolicy* (*a* # *list*) = *a* ⊕(*list2FWpolicy list*)
⟨*proof*⟩

**lemma** *l2p-aux2*[*rule-format*]: *list* = [] ⟹ *list2FWpolicy* (*a* # *list*) = *a*
⟨*proof*⟩

**lemma** *aux7aa*: ⟦*AllowPortFromTo a b poo* ∈ *set p*;
*allNetsDistinct* ((*AllowPortFromTo c d po*) # *p*); *a* ≠ *c* ∨ *b* ≠ *d*⟧ ⟹
*twoNetsDistinct a b c d*
⟨*proof*⟩

**lemma** *ANDConcEnd*: ⟦ *allNetsDistinct* (*xs* @ [*xa*]); *singleCombinators xs*⟧ ⟹
*allNetsDistinct xs*
⟨*proof*⟩

**lemma** *WP1ConcEnd*[*rule-format*]:
*wellformed-policy1* (*xs*@[*xa*]) ⟶ *wellformed-policy1 xs*
⟨*proof*⟩

**lemma** *NDComm*: *netsDistinct a b* = *netsDistinct b a*
⟨*proof*⟩

**lemma** *wellformed1-sorted*[*simp*]:
**assumes** *wp1*: *wellformed-policy1 p*
**shows** *wellformed-policy1* (*sort p l*)

⟨*proof*⟩

**lemma** *wellformed1-sortedQ*[*simp*]:
**assumes** *wp1*: *wellformed-policy1 p*
**shows** *wellformed-policy1* (*qsort p l*)
⟨*proof*⟩

**lemma** *SC1*[*simp*]: *singleCombinators p* ⟹*singleCombinators* (*removeShadowRules1 p*)
⟨*proof*⟩

**lemma** *SC2*[*simp*]: *singleCombinators p* ⟹*singleCombinators* (*removeShadowRules2 p*)
⟨*proof*⟩

**lemma** *SC3*[*simp*]: *singleCombinators p* ⟹ *singleCombinators* (*sort p l*)
⟨*proof*⟩

**lemma** *SC3Q*[*simp*]: *singleCombinators p* ⟹ *singleCombinators* (*qsort p l*)
⟨*proof*⟩

**lemma** *aND-RS1*[*simp*]: ⟦*singleCombinators p*; *allNetsDistinct p*⟧ ⟹
*allNetsDistinct* (*removeShadowRules1 p*)
⟨*proof*⟩

**lemma** *aND-RS2*[*simp*]: ⟦*singleCombinators p*; *allNetsDistinct p*⟧ ⟹
*allNetsDistinct* (*removeShadowRules2 p*)
⟨*proof*⟩

**lemma** *aND-sort*[*simp*]: ⟦*singleCombinators p*; *allNetsDistinct p*⟧ ⟹
*allNetsDistinct* (*sort p l*)
⟨*proof*⟩

**lemma** *aND-sortQ*[*simp*]: ⟦*singleCombinators p*; *allNetsDistinct p*⟧ ⟹
*allNetsDistinct* (*qsort p l*)
⟨*proof*⟩

**lemma** *inRS2*[*rule-format*,*simp*]: *x* ∉ *set p* ⟶ *x* ∉ *set* (*removeShadowRules2 p*)
⟨*proof*⟩

**lemma** *distinct-RS2*[*rule-format*,*simp*]: *distinct p* $\longrightarrow$
*distinct* (*removeShadowRules2 p*)
⟨*proof*⟩

**lemma** *setPaireq*: $\{x, y\} = \{a, b\} \Longrightarrow x = a \wedge y = b \vee x = b \wedge y = a$
⟨*proof*⟩

**lemma** *position-positive*[*rule-format*]: *a* ∈ *set l* $\longrightarrow$ *position a l* > *0*
⟨*proof*⟩

**lemma** *pos-noteq*[*rule-format*]:
*a* ∈ *set l* $\longrightarrow$ *b* ∈ *set l* $\longrightarrow$ *c* ∈ *set l* $\longrightarrow$ *a* ≠ *b* $\longrightarrow$
(*position a l*) <= (*position b l*) $\longrightarrow$
(*position b l*) <= (*position c l*) $\longrightarrow$
*a* ≠ *c*
⟨*proof*⟩

**lemma** *setPair-noteq*: {*a*,*b*} ≠ {*c*,*d*} $\Longrightarrow$ ¬ ((*a* = *c*) ∧ (*b* = *d*))
⟨*proof*⟩

**lemma** *setPair-noteq-allow*: {*a*,*b*} ≠ {*c*,*d*} $\Longrightarrow$ ¬ ((*a* = *c*) ∧ (*b* = *d*) ∧ *P*)
⟨*proof*⟩

**lemma** *order-trans*:
⟦*in-list x l*; *in-list y l*; *in-list z l*; *singleCombinators* [*x*];
*singleCombinators* [*y*]; *singleCombinators* [*z*]; *smaller x y l*; *smaller y z l*⟧ $\Longrightarrow$
*smaller x z l*
⟨*proof*⟩

**lemma** *sortedConcStart*[*rule-format*]:
*sorted* (*a* # *aa* # *p*) *l* $\longrightarrow$ *in-list a l* $\longrightarrow$ *in-list aa l* $\longrightarrow$ *all-in-list p l*$\longrightarrow$
*singleCombinators* [*a*] $\longrightarrow$ *singleCombinators* [*aa*] $\longrightarrow$ *singleCombinators p* $\longrightarrow$
*sorted* (*a*#*p*) *l*
⟨*proof*⟩

**lemma** *singleCombinatorsStart*[*simp*]: *singleCombinators* (*x*#*xs*) $\Longrightarrow$
*singleCombinators* [*x*]
⟨*proof*⟩

**lemma** *sorted-is-smaller*[*rule-format*]:
*sorted* (*a* # *p*) *l* $\longrightarrow$ *in-list a l* $\longrightarrow$ *in-list b l* $\longrightarrow$ *all-in-list p l* $\longrightarrow$
*singleCombinators* [*a*] $\longrightarrow$ *singleCombinators p* $\longrightarrow$ *b* ∈ *set p* $\longrightarrow$ *smaller a b l*
⟨*proof*⟩

**lemma** *sortedConcEnd*[*rule-format*]: *sorted* (*a* # *p*) *l* $\longrightarrow$ *in-list a l* $\longrightarrow$
*all-in-list p l* $\longrightarrow$ *singleCombinators* [*a*] $\longrightarrow$
*singleCombinators p* $\longrightarrow$ *sorted p l*
⟨*proof*⟩

**lemma** *in-set-in-list*[*rule-format*]: $a \in set\ p \longrightarrow all\text{-}in\text{-}list\ p\ l \longrightarrow in\text{-}list\ a\ l$
⟨*proof*⟩

**lemma** *sorted-Consb*[*rule-format*]:
$all\text{-}in\text{-}list\ (x\#xs)\ l \longrightarrow singleCombinators\ (x\#xs) \longrightarrow$
$(sorted\ xs\ l\ \&\ (ALL\ y{:}set\ xs.\ smaller\ x\ y\ l)) \longrightarrow\ (sorted\ (x\#xs)\ l)$
⟨*proof*⟩

**lemma** *sorted-Cons*: $[\![all\text{-}in\text{-}list\ (x\#xs)\ l;\ singleCombinators\ (x\#xs)]\!] \Longrightarrow$
$(sorted\ xs\ l\ \&\ (ALL\ y{:}set\ xs.\ smaller\ x\ y\ l)) =\ (sorted\ (x\#xs)\ l)$
⟨*proof*⟩

**lemma** *smaller-antisym*: $[\![\neg\ smaller\ a\ b\ l;\ in\text{-}list\ a\ l;\ in\text{-}list\ b\ l;$
$singleCombinators[a];\ singleCombinators\ [b]]\!] \Longrightarrow$
$smaller\ b\ a\ l$
⟨*proof*⟩

**lemma** *set-insort-insert*: $set\ (insort\ x\ xs\ l) \subseteq insert\ x\ (set\ xs)$
⟨*proof*⟩

**lemma** *all-in-listSubset*[*rule-format*]: $all\text{-}in\text{-}list\ b\ l \longrightarrow singleCombinators\ a \longrightarrow$
$set\ a \subseteq set\ b \longrightarrow all\text{-}in\text{-}list\ a\ l$
⟨*proof*⟩

**lemma** *singleCombinators-insort*: $[\![singleCombinators\ [x];\ singleCombinators\ xs]\!] \Longrightarrow$
$singleCombinators\ (insort\ x\ xs\ l)$
⟨*proof*⟩

**lemma** *all-in-list-insort*: $[\![all\text{-}in\text{-}list\ xs\ l;\ singleCombinators\ (x\#xs);$
$in\text{-}list\ x\ l]\!] \Longrightarrow\ all\text{-}in\text{-}list\ (insort\ x\ xs\ l)\ l$
⟨*proof*⟩

**lemma** *sorted-ConsA*:$[\![all\text{-}in\text{-}list\ (x\#xs)\ l;\ singleCombinators\ (x\#xs)]\!] \Longrightarrow$
$(sorted\ (x\#xs)\ l)\ = (sorted\ xs\ l\ \&\ (ALL\ y{:}set\ xs.\ smaller\ x\ y\ l))$
⟨*proof*⟩

**lemma** *is-in-insort*: $y \in set\ xs \Longrightarrow y \in set\ (insort\ x\ xs\ l)$
⟨*proof*⟩

**lemma** *sorted-insorta*[*rule-format*]:
$sorted\ (insort\ x\ xs\ l)\ l \longrightarrow all\text{-}in\text{-}list\ (x\#xs)\ l \longrightarrow distinct\ (x\#xs) \longrightarrow$
$singleCombinators\ [x] \longrightarrow singleCombinators\ xs \longrightarrow sorted\ xs\ l$
⟨*proof*⟩

**lemma** *sorted-insortb*[*rule-format*]:
$sorted\ xs\ l \longrightarrow all\text{-}in\text{-}list\ (x\#xs)\ l \longrightarrow distinct\ (x\#xs) \longrightarrow$
$singleCombinators\ [x] \longrightarrow singleCombinators\ xs \longrightarrow sorted\ (insort\ x\ xs\ l)\ l$
⟨*proof*⟩

**lemma** *sorted-insort*: ⟦*all-in-list* (*x*#*xs*) *l*; *distinct*(*x*#*xs*); *singleCombinators* [*x*];
*singleCombinators xs*⟧ ⟹
*sorted* (*insort x xs l*) *l* = *sorted xs l*
⟨*proof*⟩

**lemma** *distinct-insort*: *distinct* (*insort x xs l*) = (*x* ∉ *set xs* ∧ *distinct xs*)
⟨*proof*⟩

**lemma** *distinct-sort*[*simp*]: *distinct* (*sort xs l*) = *distinct xs*
⟨*proof*⟩

**lemma** *sort-is-sorted*[*rule-format*]: *all-in-list p l* ⟶ *distinct p* ⟶
*singleCombinators p* ⟶ *sorted* (*sort p l*) *l*
⟨*proof*⟩

**lemma** *smaller-sym*[*rule-format*]: *all-in-list* [*a*] *l* ⟶ *smaller a a l*
⟨*proof*⟩

**lemma** *SC-sublist*[*rule-format*]: *singleCombinators xs* ⟹
*singleCombinators* (*qsort* [*y*←*xs*. *P y*] *l*)
⟨*proof*⟩

**lemma** *all-in-list-sublist*[*rule-format*]: *singleCombinators xs* ⟶ *all-in-list xs l* ⟶
*all-in-list* (*qsort* [*y*←*xs*. *P y*] *l*) *l*
⟨*proof*⟩

**lemma** *SC-sublist2*[*rule-format*]: *singleCombinators xs* ⟶
*singleCombinators* ([*y*←*xs*. *P y*])
⟨*proof*⟩

**lemma** *all-in-list-sublist2*[*rule-format*]: *singleCombinators xs* ⟶ *all-in-list xs l* ⟶
*all-in-list* ( [*y*←*xs*. *P y*]) *l*
⟨*proof*⟩

**lemma** *all-in-listAppend*[*rule-format*]:
*all-in-list* (*xs*) *l* ⟶ *all-in-list* (*ys*) *l* ⟶ *all-in-list* (*xs* @ *ys*) *l*
⟨*proof*⟩

**lemma** *distinct-sortQ*[*rule-format*]: *singleCombinators xs* ⟶

*all-in-list xs l* $\longrightarrow$ *distinct xs* $\longrightarrow$ *distinct* (*qsort xs l*)
⟨*proof*⟩

**lemma** *singleCombinatorsAppend*[*rule-format*]:
*singleCombinators* (*xs*) $\longrightarrow$ *singleCombinators* (*ys*) $\longrightarrow$ *singleCombinators* (*xs* @ *ys*)
⟨*proof*⟩

**lemma** *sorted-append1*[*rule-format*]:
*all-in-list xs l* $\longrightarrow$ *singleCombinators xs* $\longrightarrow$
*all-in-list ys l* $\longrightarrow$
*singleCombinators ys* $\longrightarrow$
(*sorted* (*xs*@*ys*) *l* $\longrightarrow$ (*sorted xs l* & *sorted ys l* & ($\forall$ *x* $\in$ *set xs*. $\forall$ *y* $\in$ *set ys*. *smaller x y l*)))
⟨*proof*⟩

**lemma** *sorted-append2*[*rule-format*]:
*all-in-list xs l*$\longrightarrow$ *singleCombinators xs* $\longrightarrow$
*all-in-list ys l* $\longrightarrow$
*singleCombinators ys* $\longrightarrow$
(*sorted xs l* & *sorted ys l* & ($\forall$ *x* $\in$ *set xs*. $\forall$ *y* $\in$ *set ys*. *smaller x y l*)) $\longrightarrow$ (*sorted* (*xs*@*ys*) *l*)
⟨*proof*⟩

**lemma** *sorted-append*[*rule-format*]:
*all-in-list xs l* $\longrightarrow$ *singleCombinators xs* $\longrightarrow$
*all-in-list ys l* $\longrightarrow$
*singleCombinators ys* $\longrightarrow$ (*sorted* (*xs*@*ys*) *l*) =
(*sorted xs l* & *sorted ys l* & ($\forall$ *x* $\in$ *set xs*. $\forall$ *y* $\in$ *set ys*. *smaller x y l*))
⟨*proof*⟩

**lemma** *sort-is-sortedQ*[*rule-format*]: *all-in-list p l* $\longrightarrow$
*singleCombinators p* $\longrightarrow$ *sorted* (*qsort p l*) *l*
⟨*proof*⟩

**lemma** *inSet-not-MT*: *a* $\in$ *set p* $\Longrightarrow$ *p* $\neq$ []
⟨*proof*⟩

**lemma** *RS1n-assoc*: *x* $\neq$ *DenyAll* $\Longrightarrow$ *removeShadowRules1-alternative xs* @ [*x*] =
*removeShadowRules1-alternative* (*xs* @ [*x*])
⟨*proof*⟩

**lemma** *RS1n-nMT*[*rule-format*,*simp*]: *p* $\neq$ []$\longrightarrow$ *removeShadowRules1-alternative p* $\neq$ []
⟨*proof*⟩

**lemma** *RS1N-DA*[*simp*]: *removeShadowRules1-alternative* (*a*@[*DenyAll*]) = [*DenyAll*]
⟨*proof*⟩

**lemma** *WP1n-DA-notinSet*[*rule-format*]: *wellformed-policy1-strong p* $\longrightarrow$
*DenyAll* $\notin$ *set* (*tl p*)
⟨*proof*⟩

**lemma** *mt-sym*: *dom a* $\cap$ *dom b* = {} $\Longrightarrow$ *dom b* $\cap$ *dom a* = {}
⟨*proof*⟩

**lemma** *DAnotTL*[*rule-format*]:
*xs* $\neq$ [] $\longrightarrow$ *wellformed-policy1* (*xs* @ [*DenyAll*]) $\longrightarrow$ *False*
⟨*proof*⟩

**lemma** *AND-tl*[*rule-format*]: *allNetsDistinct* ( *p*) $\longrightarrow$ *allNetsDistinct* (*tl p*)
⟨*proof*⟩

**lemma** *distinct-tl*[*rule-format*]: *distinct p* $\longrightarrow$ *distinct* (*tl p*)
⟨*proof*⟩

**lemma** *SC-tl*[*rule-format*]: *singleCombinators* ( *p*) $\longrightarrow$ *singleCombinators* (*tl p*)
⟨*proof*⟩

**lemma** *Conc-not-MT*: *p* = *x*#*xs* $\Longrightarrow$ *p* $\neq$ []
⟨*proof*⟩

**lemma** *wp1-tl*[*rule-format*]: *p* $\neq$ [] $\wedge$ *wellformed-policy1 p* $\longrightarrow$
*wellformed-policy1* (*tl p*)
⟨*proof*⟩

**lemma** *wp1-eq*[*rule-format*]: *wellformed-policy1-strong p* $\Longrightarrow$ *wellformed-policy1 p*
⟨*proof*⟩

**lemma** *wellformed1-alternative-sorted*: *wellformed-policy1-strong p* $\Longrightarrow$
*wellformed-policy1-strong* (*sort p l*)
⟨*proof*⟩

**lemma** *wp1n-RS2*[*rule-format*]: *wellformed-policy1-strong p* $\longrightarrow$
*wellformed-policy1-strong* (*removeShadowRules2 p*)
⟨*proof*⟩

**lemma** *RS2-NMT*[*rule-format*]: *p* $\neq$ [] $\longrightarrow$ *removeShadowRules2 p* $\neq$ []
⟨*proof*⟩

**lemma** *wp1-alternative-not-mt*[*simp*]: *wellformed-policy1-strong p* $\Longrightarrow$ *p* $\neq$ []
⟨*proof*⟩

**lemma** *AIL1*[*rule-format*,*simp*]: *all-in-list p l* $\longrightarrow$
*all-in-list* (*removeShadowRules1 p*) *l*
⟨*proof*⟩

**lemma** *wp1ID*: *wellformed-policy1-strong* (*insertDeny* (*removeShadowRules1 p*))
⟨*proof*⟩

**lemma** *dRD*[*simp*]: *distinct* (*remdups p*)
⟨*proof*⟩

**lemma** *AILrd*[*rule-format*,*simp*]: *all-in-list p l* $\longrightarrow$ *all-in-list* (*remdups p*) *l*
⟨*proof*⟩

**lemma** *AILiD*[*rule-format*,*simp*]: *all-in-list p l* $\longrightarrow$ *all-in-list* (*insertDeny p*) *l*
⟨*proof*⟩

**lemma** *SCrd*[*rule-format*,*simp*]:*singleCombinators p*$\longrightarrow$ *singleCombinators*(*remdups p*)
⟨*proof*⟩

**lemma** *SCRiD*[*rule-format*,*simp*]: *singleCombinators p* $\longrightarrow$
*singleCombinators*(*insertDeny p*)
⟨*proof*⟩

**lemma** *WP1rd*[*rule-format*,*simp*]: *wellformed-policy1-strong p* $\longrightarrow$
*wellformed-policy1-strong* (*remdups p*)
⟨*proof*⟩

**lemma** *ANDrd*[*rule-format*,*simp*]: *singleCombinators p* $\longrightarrow$ *allNetsDistinct p* $\longrightarrow$
*allNetsDistinct* (*remdups p*)
⟨*proof*⟩

**lemma** *ANDiD*[*rule-format*,*simp*]: *allNetsDistinct p* $\longrightarrow$
*allNetsDistinct* (*insertDeny p*)
⟨*proof*⟩

**lemma** *mr-iD*[*rule-format*]: *wellformed-policy1-strong p* $\longrightarrow$
*matching-rule x p* = *matching-rule x* (*insertDeny p*)
⟨*proof*⟩

**lemma** *WP1iD*[*rule-format*,*simp*]: *wellformed-policy1-strong p* $\longrightarrow$

*wellformed-policy1-strong* (*insertDeny p*)
⟨*proof*⟩

**lemma** *DAiniD*: *DenyAll* ∈ *set* (*insertDeny p*)
⟨*proof*⟩

**lemma** *p2lNmt*: *policy2list p* ≠ []
⟨*proof*⟩

**lemma** *AIL2*[*rule-format*,*simp*]: *all-in-list p l* ⟶
*all-in-list* (*removeShadowRules2 p*) *l*
⟨*proof*⟩

**lemma** *SCConc*: ⟦*singleCombinators x*; *singleCombinators y*⟧ ⟹
*singleCombinators* (*x*@*y*)
⟨*proof*⟩

**lemma** *SCp2l*: *singleCombinators* (*policy2list p*)
⟨*proof*⟩

**lemma** *subnetAux*: *Dd* ∩ *A* ≠ {} ⟹ *A* ⊆ *B* ⟹ *Dd* ∩ *B* ≠ {}
⟨*proof*⟩

**lemma** *soadisj*: ⟦*x* ∈ *subnetsOfAdr a*; *y* ∈ *subnetsOfAdr a*⟧ ⟹ ¬ *netsDistinct x y*
⟨*proof*⟩

**lemma** *not-member*: ¬ *member a* (*x*⊕*y*) ⟹ ¬ *member a x*
⟨*proof*⟩

**lemma** *soadisj2*: (∀ *a x y*. *x* ∈ *subnetsOfAdr a* ∧ *y* ∈ *subnetsOfAdr a* ⟶
¬ *netsDistinct x y*)
⟨*proof*⟩

**lemma** *ndFalse1*: ⟦(∀ *a b c d*. (*a*,*b*)∈*A* ∧ (*c*,*d*)∈*B* ⟶ *netsDistinct a c*);
∃(*a*, *b*)∈*A*. *a* ∈ *subnetsOfAdr D*;
∃(*a*, *b*)∈*B*. *a* ∈ *subnetsOfAdr D*⟧
⟹ *False*
⟨*proof*⟩

**lemma** *ndFalse2*: ⟦(∀ *a b c d*. (*a*,*b*)∈*A* ∧ (*c*,*d*)∈*B* ⟶ *netsDistinct b d*);
∃(*a*, *b*)∈*A*. *b* ∈ *subnetsOfAdr D*;
∃(*a*, *b*)∈*B*. *b* ∈ *subnetsOfAdr D*⟧
⟹ *False*
⟨*proof*⟩

**lemma** *tndFalse*: ⟦(∀ *a b c d*. (*a*,*b*)∈*A* ∧ (*c*,*d*)∈*B* ⟶ *twoNetsDistinct a b c d*);

$\exists\,(a,\ b){\in}A.\ a \in subnetsOfAdr\ (D{::}('a{::}adr)) \wedge b \in subnetsOfAdr\ (F{::}'a);$
$\exists\,(a,\ b){\in}B.\ a \in subnetsOfAdr\ D \wedge\ b{\in}\ subnetsOfAdr\ F\rrbracket$
$\Longrightarrow False$
⟨*proof*⟩

**lemma** *sepnMT*[*rule-format*]: $p \neq [] \longrightarrow (separate\ p) \neq []$
⟨*proof*⟩

**lemma** *sepDA*[*rule-format*]: $DenyAll \notin set\ p \longrightarrow DenyAll \notin set\ (separate\ p)$
⟨*proof*⟩

**lemma** *setnMT*: $set\ a = set\ b \Longrightarrow a \neq [] \Longrightarrow b \neq []$
⟨*proof*⟩

**lemma** *sortnMT*: $p \neq [] \Longrightarrow sort\ p\ l \neq []$
⟨*proof*⟩

**lemma** *idNMT*[*rule-format*]: $p \neq [] \longrightarrow insertDenies\ p \neq []$
⟨*proof*⟩

**lemma** *OTNoTN*[*rule-format*]: $OnlyTwoNets\ p \longrightarrow x \neq DenyAll \longrightarrow x \in set\ p \longrightarrow$
$onlyTwoNets\ x$
⟨*proof*⟩

**lemma** *first-isIn*[*rule-format*]:
$\neg\ member\ DenyAll\ x \longrightarrow (first\text{-}srcNet\ x, first\text{-}destNet\ x) \in sdnets\ x$
⟨*proof*⟩

**lemma** *sdnets2*: $[\![\exists\ a\ b.\ sdnets\ x = \{(a,\ b),\ (b,\ a)\};\ \neg\ member\ DenyAll\ x]\!] \Longrightarrow$
$sdnets\ x = \{(first\text{-}srcNet\ x,\ first\text{-}destNet\ x),$
$(first\text{-}destNet\ x,\ first\text{-}srcNet\ x)\}$
⟨*proof*⟩

**lemma** *alternativelistconc1*[*rule-format*]: $a \in set\ (net\text{-}list\text{-}aux\ [x]) \longrightarrow$
$a \in set\ (net\text{-}list\text{-}aux\ [x,y])$
⟨*proof*⟩

**lemma** *alternativelistconc2*[*rule-format*]: $a \in set\ (net\text{-}list\text{-}aux\ [x]) \longrightarrow$
$a \in set\ (net\text{-}list\text{-}aux\ [y,x])$
⟨*proof*⟩

**lemma** *noDA*[*rule-format*]:$noDenyAll\ xs \longrightarrow s \in set\ xs \longrightarrow \neg\ member\ DenyAll\ s$
⟨*proof*⟩

**lemma** *isInAlternativeList*:
$(aa \in set\ (net\text{-}list\text{-}aux\ [a]) \vee aa \in set\ (net\text{-}list\text{-}aux\ p))$
$\Longrightarrow aa \in set\ (net\text{-}list\text{-}aux\ (a\ \#\ p))$
⟨*proof*⟩

**lemma** *netlistaux*: $x \in set\ (net\text{-}list\text{-}aux\ (a\ \#\ p)) \Longrightarrow$
$x \in set\ (net\text{-}list\text{-}aux\ ([a])) \vee x \in set\ (net\text{-}list\text{-}aux\ (p))$
⟨*proof*⟩

**lemma** *firstInNet*[*rule-format*]: $\neg\ member\ DenyAll\ a \longrightarrow$
$first\text{-}destNet\ a \in set\ (net\text{-}list\text{-}aux\ (a\ \#\ p))$
⟨*proof*⟩

**lemma** *firstInNeta*[*rule-format*]: $\neg\ member\ DenyAll\ a \longrightarrow$
$first\text{-}srcNet\ a \in set\ (net\text{-}list\text{-}aux\ (a\ \#\ p))$
⟨*proof*⟩

**lemma** *disjComm*: $disjSD\text{-}2\ a\ b \Longrightarrow disjSD\text{-}2\ b\ a$
⟨*proof*⟩

**lemma** *disjSD2aux*: $[\![disjSD\text{-}2\ a\ b; \neg\ member\ DenyAll\ a; \neg\ member\ DenyAll\ b]\!] \Longrightarrow$
$disjSD\text{-}2\ (DenyAllFromTo\ (first\text{-}srcNet\ a)\ (first\text{-}destNet\ a) \oplus$
$DenyAllFromTo\ (first\text{-}destNet\ a)\ (first\text{-}srcNet\ a) \oplus a)\ \ b$
⟨*proof*⟩

**lemma** *noDA1eq*[*rule-format*]: $noDenyAll\ p \longrightarrow noDenyAll1\ p$
⟨*proof*⟩

**lemma** *noDA1C*[*rule-format*]: $noDenyAll1\ (a\#p) \longrightarrow noDenyAll1\ p$
⟨*proof*⟩

**lemma** *disjSD-2IDa*: $[\![disjSD\text{-}2\ x\ y; \neg\ member\ DenyAll\ x; \neg\ member\ DenyAll\ y;$
$a = (first\text{-}srcNet\ x); b = (first\text{-}destNet\ x)]\!] \Longrightarrow$
$disjSD\text{-}2\ ((DenyAllFromTo\ a\ b) \oplus (DenyAllFromTo\ b\ a) \oplus x)\ y$
⟨*proof*⟩

**lemma** *noDAID*[*rule-format*]: $noDenyAll\ p \longrightarrow noDenyAll\ (insertDenies\ p)$
⟨*proof*⟩

**lemma** *isInIDo*[*rule-format*]: $noDenyAll\ p \longrightarrow s \in set\ (insertDenies\ p) \longrightarrow$
$(\exists!\ a.\ s = (DenyAllFromTo\ (first\text{-}srcNet\ a)\ (first\text{-}destNet\ a)) \oplus$
$(DenyAllFromTo\ (first\text{-}destNet\ a)\ (first\text{-}srcNet\ a)) \oplus a \wedge a \in set\ p)$
⟨*proof*⟩

**lemma** *id-aux1*[*rule-format*]: $DenyAllFromTo\ (first\text{-}srcNet\ s)\ (first\text{-}destNet\ s) \oplus$
$DenyAllFromTo\ (first\text{-}destNet\ s)\ (first\text{-}srcNet\ s) \oplus s \in set\ (insertDenies\ p)$
$\longrightarrow s \in set\ p$
⟨*proof*⟩

**lemma** *id-aux2*:
$[\![noDenyAll\ p; (\forall\ s.\ s \in set\ p \longrightarrow disjSD\text{-}2\ a\ s); \neg\ member\ DenyAll\ a;$
$((DenyAllFromTo\ (first\text{-}srcNet\ s)\ (first\text{-}destNet\ s)) \oplus (DenyAllFromTo$

$(first\text{-}destNet\ s)\ (first\text{-}srcNet\ s)) \oplus s) \in set\ (insertDenies\ p)[\!] \Longrightarrow$
$disjSD\text{-}2\ a\ ((DenyAllFromTo\ (first\text{-}srcNet\ s)\ (first\text{-}destNet\ s)) \oplus$
$(DenyAllFromTo\ (first\text{-}destNet\ s)\ (first\text{-}srcNet\ s)) \oplus s)$
⟨*proof*⟩

**lemma** *id-aux4*[*rule-format*]: $[\![noDenyAll\ p;\ (\forall s.\ s \in set\ p \longrightarrow$
$disjSD\text{-}2\ a\ s); s \in set\ (insertDenies\ p);\ \neg\ member\ DenyAll\ a]\!] \Longrightarrow\ disjSD\text{-}2\ a\ s$
⟨*proof*⟩

**lemma** *sepNetsID*[*rule-format*]: $noDenyAll1\ p \longrightarrow separated\ p \longrightarrow$
$separated\ (insertDenies\ p)$
⟨*proof*⟩

**lemma** *aNDDA*[*rule-format*]: $allNetsDistinct\ p \longrightarrow allNetsDistinct(DenyAll\#p)$
⟨*proof*⟩

**lemma** *OTNConc*[*rule-format*]: $OnlyTwoNets\ (y\ \#\ z) \longrightarrow\ OnlyTwoNets\ z$
⟨*proof*⟩

**lemma** *first-bothNetsd*: $\neg\ member\ DenyAll\ x \Longrightarrow$
$first\text{-}bothNet\ x = \{first\text{-}srcNet\ x,\ first\text{-}destNet\ x\}$
⟨*proof*⟩

**lemma** *bNaux*:
$[\![\neg member\ DenyAll\ x;\ \neg\ member\ DenyAll\ y; first\text{-}bothNet\ x = first\text{-}bothNet\ y]\!]$
$\Longrightarrow \{first\text{-}srcNet\ x,\ first\text{-}destNet\ x\} = \{first\text{-}srcNet\ y,\ first\text{-}destNet\ y\}$
⟨*proof*⟩

**lemma** *setPair*: $\{a,b\} = \{a,d\} \Longrightarrow b = d$
⟨*proof*⟩

**lemma** *setPair1*: $\{a,b\} = \{d,a\} \Longrightarrow b = d$
⟨*proof*⟩

**lemma** *setPair4*: $\{a,b\} = \{c,d\} \Longrightarrow a \neq c \Longrightarrow a = d$
⟨*proof*⟩

**lemma** *otnaux1*: $\{x,\ y,\ x,\ y\} = \{x,y\}$
⟨*proof*⟩

**lemma** *OTNIDaux4*: $\{x,y,x\} = \{y,x\}$
⟨*proof*⟩

**lemma** *setPair5*: $\{a,b\} = \{c,d\} \Longrightarrow a \neq c \Longrightarrow a = d$
⟨*proof*⟩

**lemma** *otnaux*:

$[\![first\text{-}bothNet\ x = first\text{-}bothNet\ y;\ \neg\ member\ DenyAll\ x;\ \neg\ member\ DenyAll\ y;$
$onlyTwoNets\ y;\ onlyTwoNets\ x]\!] \Longrightarrow$
$onlyTwoNets\ (x \oplus y)$
⟨*proof*⟩

**lemma** *OTNSepaux*: $[\![onlyTwoNets\ (a \oplus y) \wedge OnlyTwoNets\ z \longrightarrow$
$OnlyTwoNets\ (separate\ (a \oplus y\ \#\ z));$
$\neg\ member\ DenyAll\ a;$
$\neg\ member\ DenyAll\ y;\ noDenyAll\ z;$
$onlyTwoNets\ a;\ OnlyTwoNets\ (y\ \#\ z);first\text{-}bothNet\ (a) = first\text{-}bothNet\ y]\!]$
$\Longrightarrow OnlyTwoNets\ (separate\ (a \oplus y\ \#\ z))$
⟨*proof*⟩

**lemma** *OTNSEp*[*rule-format*]: $noDenyAll1\ p \longrightarrow OnlyTwoNets\ p \longrightarrow$
$OnlyTwoNets\ (separate\ p)$
⟨*proof*⟩

**lemma** *nda*[*rule-format*]: $singleCombinators\ (a\#p) \longrightarrow noDenyAll\ p \longrightarrow$
$noDenyAll1\ (a\ \#\ p)$
⟨*proof*⟩

**lemma** *nDAcharn*[*rule-format*]: $noDenyAll\ p = (\forall\ r \in set\ p.\ \neg\ member\ DenyAll\ r)$
⟨*proof*⟩

**lemma** *nDAeqSet*: $set\ p = set\ s \Longrightarrow noDenyAll\ p = noDenyAll\ s$
⟨*proof*⟩

**lemma** *nDASCaux*[*rule-format*]: $DenyAll \notin set\ p \longrightarrow singleCombinators\ p \longrightarrow$
$r \in set\ p \longrightarrow\ \neg\ member\ DenyAll\ r$
⟨*proof*⟩

**lemma** *nDASC*[*rule-format*]: $wellformed\text{-}policy1\ p \longrightarrow singleCombinators\ p \longrightarrow$
$noDenyAll1\ p$
⟨*proof*⟩

**lemma** *noDAAll*[*rule-format*]: $noDenyAll\ p = (\neg\ memberP\ DenyAll\ p)$
⟨*proof*⟩

**lemma** *memberPsep*[*symmetric*]: $memberP\ x\ p = memberP\ x\ (separate\ p)$
⟨*proof*⟩

**lemma** *noDAsep*[*rule-format*]: $noDenyAll\ p \Longrightarrow noDenyAll\ (separate\ p)$
⟨*proof*⟩

**lemma** *noDA1sep*[*rule-format*]: $noDenyAll1\ p \longrightarrow noDenyAll1\ (separate\ p)$
⟨*proof*⟩

**lemma** *isInAlternativeLista*: $(aa \in set\ (net\text{-}list\text{-}aux\ [a])) \Longrightarrow$
$aa \in set\ (net\text{-}list\text{-}aux\ (a\ \#\ p))$

⟨*proof*⟩

**lemma** *isInAlternativeListb*: (*aa* ∈ *set* (*net-list-aux p*))⟹
*aa* ∈ *set* (*net-list-aux* (*a* # *p*))
⟨*proof*⟩

**lemma** *ANDSepaux*: *allNetsDistinct* (*x* # *y* # *z*) ⟹ *allNetsDistinct* (*x* ⊕ *y* # *z*)
⟨*proof*⟩

**lemma** *netlistalternativeSeparateaux*:
*net-list-aux* [*y*] @ *net-list-aux z* = *net-list-aux* (*y* # *z*)
⟨*proof*⟩

**lemma** *netlistalternativeSeparate*: *net-list-aux p* = *net-list-aux* (*separate p*)
⟨*proof*⟩

**lemma** *ANDSepaux2*: ⟦*allNetsDistinct* (*x* # *y* # *z*);
*allNetsDistinct* (*separate* (*y* # *z*))⟧
⟹ *allNetsDistinct* (*x* # *separate* (*y* # *z*))
⟨*proof*⟩

**lemma** *ANDSep*[*rule-format*]: *allNetsDistinct p* ⟶ *allNetsDistinct*(*separate p*)
⟨*proof*⟩

**lemma** *wp1-alternativesep*[*rule-format*]: *wellformed-policy1-strong p* ⟶
*wellformed-policy1-strong* (*separate p*)
⟨*proof*⟩

**lemma** *noDAsort*[*rule-format*]: *noDenyAll1 p* ⟶ *noDenyAll1* (*sort p l*)
⟨*proof*⟩

**lemma** *OTNSC*[*rule-format*]: *singleCombinators p* ⟶ *OnlyTwoNets p*
⟨*proof*⟩

**lemma** *fMTaux*: ¬ *member DenyAll x* ⟹ *first-bothNet x* ≠ {}
⟨*proof*⟩

**lemma** *fl2*[*rule-format*]: *firstList* (*separate p*) = *firstList p*
⟨*proof*⟩

**lemma** *fl3*[*rule-format*]: *NetsCollected p* ⟶ (*first-bothNet x* ≠ *firstList p* ⟶
(∀ *a*∈*set p*. *first-bothNet x* ≠ *first-bothNet a*))⟶ *NetsCollected* (*x*#*p*)
⟨*proof*⟩

**lemma** *sortedConc*[*rule-format*]: *sorted* (*a* # *p*) *l* ⟶ *sorted p l*
⟨*proof*⟩

**lemma** *smalleraux2*:
$\{a,b\} \in set\ l \Longrightarrow \{c,d\} \in set\ l \Longrightarrow \{a,b\} \neq \{c,d\} \Longrightarrow$
$smaller\ (DenyAllFromTo\ a\ b)\ (DenyAllFromTo\ c\ d)\ l \Longrightarrow$
$\neg\ smaller\ (DenyAllFromTo\ c\ d)\ (DenyAllFromTo\ a\ b)\ l$
⟨*proof*⟩

**lemma** *smalleraux2a*:
$\{a,b\} \in set\ l \Longrightarrow \{c,d\} \in set\ l \Longrightarrow \{a,b\} \neq \{c,d\} \Longrightarrow$
$smaller\ (DenyAllFromTo\ a\ b)\ (AllowPortFromTo\ c\ d\ p)\ l \Longrightarrow$
$\neg\ smaller\ (AllowPortFromTo\ c\ d\ p)\ (DenyAllFromTo\ a\ b)\ l$
⟨*proof*⟩

**lemma** *smalleraux2b*:
$\{a,b\} \in set\ l \Longrightarrow \{c,d\} \in set\ l \Longrightarrow \{a,b\} \neq \{c,d\} \Longrightarrow y = DenyAllFromTo\ a\ b \Longrightarrow$
$smaller\ (AllowPortFromTo\ \ c\ d\ p)\ y\ l \Longrightarrow$
$\neg\ smaller\ y\ (AllowPortFromTo\ \ c\ d\ p)\ l$
⟨*proof*⟩

**lemma** *smalleraux2c*:
$\{a,b\} \in set\ l \Longrightarrow \{c,d\} \in set\ l \Longrightarrow \{a,b\} \neq \{c,d\} \Longrightarrow y = AllowPortFromTo\ a\ b\ q \Longrightarrow$
$smaller\ (AllowPortFromTo\ \ c\ d\ p)\ y\ l \Longrightarrow \neg\ smaller\ y\ (AllowPortFromTo\ \ c\ d\ p)\ l$
⟨*proof*⟩

**lemma** *smalleraux3*:
**assumes** $x \in set\ l$
**assumes** $y \in set\ l$
**assumes** $x \neq y$
**assumes** $x = bothNet\ a$
**assumes** $y = bothNet\ b$
**assumes** $smaller\ a\ b\ l$
**assumes** $singleCombinators\ [a]$
**assumes** $singleCombinators\ [b]$
**shows** $\neg\ smaller\ b\ a\ l$
⟨*proof*⟩

**lemma** *smalleraux3a*:
$a \neq DenyAll \Longrightarrow b \neq DenyAll \Longrightarrow in\text{-}list\ b\ l \Longrightarrow in\text{-}list\ a\ l \Longrightarrow$
$bothNet\ a \neq bothNet\ b \Longrightarrow smaller\ a\ b\ l \Longrightarrow singleCombinators\ [a] \Longrightarrow$
$singleCombinators\ [b] \Longrightarrow \neg\ smaller\ b\ a\ l$
⟨*proof*⟩

**lemma** *posaux*[*rule-format*]: $position\ a\ l < position\ b\ l \longrightarrow a \neq b$
⟨*proof*⟩

**lemma** *posaux6*[*rule-format*]: $a \in set\ l \longrightarrow b \in set\ l \longrightarrow a \neq b \longrightarrow$
$position\ a\ l \neq\ position\ b\ l$
⟨*proof*⟩

**lemma** *notSmallerTransaux*[*rule-format*]:
$[\![x \neq DenyAll;\ r \neq DenyAll;\ singleCombinators\ [x];\ singleCombinators\ [y];$
$singleCombinators\ [r];\ \neg\ smaller\ y\ x\ l;\ smaller\ x\ y\ l;\ smaller\ x\ r\ l;$
$smaller\ y\ r\ l;\ in\text{-}list\ x\ l;\ in\text{-}list\ y\ l;\ in\text{-}list\ r\ l]\!] \Longrightarrow$
$\neg\ smaller\ r\ x\ l$
⟨*proof*⟩

**lemma** *notSmallerTrans*[*rule-format*]:
$x \neq DenyAll \longrightarrow r \neq DenyAll \longrightarrow singleCombinators\ (x\#y\#z) \longrightarrow$
$\neg\ smaller\ y\ x\ l \longrightarrow sorted\ (x\#y\#z)\ l \longrightarrow r \in set\ z \longrightarrow$
$all\text{-}in\text{-}list\ (x\#y\#z)\ l \longrightarrow \neg\ smaller\ r\ x\ l$
⟨*proof*⟩

**lemma** *NCSaux1*[*rule-format*]:
$noDenyAll\ p \longrightarrow \{x,\ y\} \in set\ l \longrightarrow\ all\text{-}in\text{-}list\ p\ l \longrightarrow singleCombinators\ p \longrightarrow$
$sorted\ (DenyAllFromTo\ x\ y\ \#\ p)\ l \longrightarrow \{x,\ y\} \neq firstList\ p \longrightarrow$
$DenyAllFromTo\ u\ v \in set\ p \longrightarrow \{x,\ y\} \neq \{u,\ v\}$
⟨*proof*⟩

**lemma** *posaux3*[*rule-format*]:
$a \in set\ l \longrightarrow b \in set\ l \longrightarrow a \neq b \longrightarrow position\ a\ l \neq position\ b\ l$
⟨*proof*⟩

**lemma** *posaux4*[*rule-format*]: $singleCombinators\ [a] \longrightarrow a \neq DenyAll \longrightarrow$
$b \neq DenyAll \longrightarrow in\text{-}list\ a\ l \longrightarrow in\text{-}list\ b\ l \longrightarrow$
$smaller\ a\ b\ \ l \longrightarrow x = (bothNet\ a) \longrightarrow$
$y = (bothNet\ b) \longrightarrow position\ x\ l <= position\ y\ l$
⟨*proof*⟩

**lemma** *NCSaux2*[*rule-format*]:
$noDenyAll\ p \longrightarrow \{a,\ b\} \in set\ l \longrightarrow all\text{-}in\text{-}list\ p\ l \longrightarrow singleCombinators\ p \longrightarrow$
$sorted\ (DenyAllFromTo\ a\ b\ \#\ p)\ l \longrightarrow \{a,\ b\} \neq firstList\ p \longrightarrow$
$AllowPortFromTo\ u\ v\ w \in set\ p \longrightarrow\ \{a,\ b\} \neq \{u,\ v\}$
⟨*proof*⟩

**lemma** *NCSaux3*[*rule-format*]:
$noDenyAll\ p \longrightarrow \{a,\ b\} \in set\ l \longrightarrow\ all\text{-}in\text{-}list\ p\ l \longrightarrow singleCombinators\ p \longrightarrow$
$sorted\ (AllowPortFromTo\ a\ b\ w\ \#\ p)\ l \longrightarrow \{a,\ b\} \neq firstList\ p \longrightarrow$
$DenyAllFromTo\ u\ v \in set\ p \longrightarrow \{a,\ b\} \neq \{u,\ v\}$
⟨*proof*⟩

**lemma** *NCSaux4*[*rule-format*]:
$noDenyAll\ p \longrightarrow \{a,\ b\} \in set\ l \longrightarrow\ all\text{-}in\text{-}list\ p\ l \longrightarrow singleCombinators\ p \longrightarrow$
$sorted\ (AllowPortFromTo\ a\ b\ c\ \#\ p)\ l \longrightarrow \{a,\ b\} \neq firstList\ p \longrightarrow$
$AllowPortFromTo\ u\ v\ w \in set\ p \longrightarrow \{a,\ b\} \neq \{u,\ v\}$
⟨*proof*⟩

**lemma** *NetsCollectedSorted*[*rule-format*]:
*noDenyAll1 p* $\longrightarrow$ *all-in-list p l* $\longrightarrow$ *singleCombinators p* $\longrightarrow$ *sorted p l* $\longrightarrow$
*NetsCollected p*
⟨*proof*⟩

**lemma** *NetsCollectedSort*: *distinct p* $\Longrightarrow$*noDenyAll1 p* $\Longrightarrow$ *all-in-list p l* $\Longrightarrow$
*singleCombinators p* $\Longrightarrow$ *NetsCollected* (*sort p l*)
⟨*proof*⟩

**lemma** *fBNsep*[*rule-format*]: ($\forall$ *a*$\in$*set z*. {*b*,*c*} $\neq$ *first-bothNet a*) $\longrightarrow$
($\forall$ *a*$\in$*set* (*separate z*). {*b*,*c*} $\neq$ *first-bothNet a*)
⟨*proof*⟩

**lemma** *fBNsep1*[*rule-format*]: ($\forall$ *a*$\in$*set z*. *first-bothNet x* $\neq$ *first-bothNet a*) $\longrightarrow$
($\forall$ *a*$\in$*set* (*separate z*). *first-bothNet x* $\neq$ *first-bothNet a*)
⟨*proof*⟩

**lemma** *NetsCollectedSepauxa*:
⟦{*b*,*c*} $\neq$ *firstList z*; *noDenyAll1 z*;
($\forall$ *a*$\in$*set z*. {*b*,*c*} $\neq$ *first-bothNet a*); *NetsCollected* (*z*);
*NetsCollected* (*separate* (*z*)); {*b*,*c*} $\neq$ *firstList* (*separate* (*z*));
*a* $\in$ *set* (*separate* (*z*))⟧ $\Longrightarrow$
{*b*,*c*} $\neq$ *first-bothNet a*
⟨*proof*⟩

**lemma** *NetsCollectedSepaux*:
⟦*first-bothNet* (*x*::('*a*,'*b*)*Combinators*) $\neq$ *first-bothNet y*; $\neg$ *member DenyAll y* $\wedge$
*noDenyAll z*;
($\forall$ *a*$\in$*set z*. *first-bothNet x* $\neq$ *first-bothNet a*) $\wedge$ *NetsCollected* (*y* # *z*);
*NetsCollected* (*separate* (*y* # *z*)); *first-bothNet x* $\neq$ *firstList* (*separate* (*y* # *z*));
*a* $\in$ *set* (*separate* (*y* # *z*))⟧ $\Longrightarrow$
*first-bothNet* (*x*::('*a*,'*b*)*Combinators*) $\neq$ *first-bothNet* (*a*::('*a*,'*b*)*Combinators*)
⟨*proof*⟩

**lemma** *NetsCollectedSep*[*rule-format*]: *noDenyAll1 p* $\longrightarrow$ *NetsCollected p* $\longrightarrow$
*NetsCollected* (*separate p*)
⟨*proof*⟩

**lemma** *OTNaux*:

*onlyTwoNets a* $\Longrightarrow \neg$ *member DenyAll a* $\Longrightarrow$ $(x,y) \in$ *sdnets a* $\Longrightarrow$
$(x =$ *first-srcNet a* $\wedge$ $y =$ *first-destNet a*$) \vee$
$(x =$ *first-destNet a* $\wedge$ $y =$ *first-srcNet a*$)$
⟨*proof*⟩

**lemma** *sdnets-charn*: *onlyTwoNets a* $\Longrightarrow \neg$ *member DenyAll a* $\Longrightarrow$
*sdnets a* $= \{$(*first-srcNet a*,*first-destNet a*)$\} \vee$
*sdnets a* $= \{$(*first-srcNet a*, *first-destNet a*),(*first-destNet a*, *first-srcNet a*)$\}$
⟨*proof*⟩

**lemma** *first-bothNet-charn*[*rule-format*]: $\neg$ *member DenyAll a* $\longrightarrow$
*first-bothNet a* $= \{$*first-srcNet a*, *first-destNet a*$\}$
⟨*proof*⟩

**lemma** *sdnets-noteq*:
$[\![$*onlyTwoNets a*; *onlyTwoNets aa*; *first-bothNet a* $\neq$ *first-bothNet aa*;
$\neg$ *member DenyAll a*; $\neg$ *member DenyAll aa*$]\!]$
$\Longrightarrow$ *sdnets a* $\neq$ *sdnets aa*
⟨*proof*⟩

**lemma** *fbn-noteq*:
$[\![$*onlyTwoNets a*; *onlyTwoNets aa*; *first-bothNet a* $\neq$ *first-bothNet aa*;
$\neg$ *member DenyAll a*; $\neg$ *member DenyAll aa*; *allNetsDistinct* $[a, aa]]\!] \Longrightarrow$
*first-srcNet a* $\neq$ *first-srcNet aa* $\vee$ *first-srcNet a* $\neq$ *first-destNet aa* $\vee$
*first-destNet a* $\neq$ *first-srcNet aa* $\vee$ *first-destNet a* $\neq$ *first-destNet aa*
⟨*proof*⟩

**lemma** *NCisSD2aux*:
$[\![$*onlyTwoNets a*; *onlyTwoNets aa*; *first-bothNet a* $\neq$ *first-bothNet aa*;
$\neg$ *member DenyAll a*; $\neg$ *member DenyAll aa*; *allNetsDistinct* $[a, aa]]\!] \Longrightarrow$
*disjSD-2 a aa*
⟨*proof*⟩

**lemma** *ANDaux3*[*rule-format*]: $y \in$ *set xs* $\longrightarrow$ $a \in$ *set* (*net-list-aux* $[y]) \longrightarrow$
$a \in$ *set* (*net-list-aux xs*)
⟨*proof*⟩

**lemma** *ANDaux2*: *allNetsDistinct* $(x \#$ *xs*$) \Longrightarrow y \in$ *set xs*
$\Longrightarrow$ *allNetsDistinct* $[x,y]$
⟨*proof*⟩

**lemma** *NCisSD2*[*rule-format*]:
$[\![\neg$ *member DenyAll a*; *OnlyTwoNets* $(a\#p)$; *NetsCollected2* $(a \#$ *p*$)$;
*NetsCollected* $(a\#p)$;*noDenyAll* $(p)$; *allNetsDistinct* $(a \#$ *p*$)$; $s \in$ *set p*$]\!] \Longrightarrow$
*disjSD-2 a s*
⟨*proof*⟩

**lemma** *separatedNC*[*rule-format*]:

*OnlyTwoNets p* ⟶ *NetsCollected2 p* ⟶ *NetsCollected p* ⟶ *noDenyAll1 p* ⟶
*allNetsDistinct p* ⟶ *separated p*
⟨*proof*⟩

**lemma** *NC2Sep*[*rule-format*]: *noDenyAll1 p* ⟶ *NetsCollected2* (*separate p*)
⟨*proof*⟩

**lemma** *separatedSep*[*rule-format*]:
*OnlyTwoNets p* ⟶ *NetsCollected2 p* ⟶ *NetsCollected p* ⟶ *noDenyAll1 p* ⟶
*allNetsDistinct p* ⟶ *separated* (*separate p*)
⟨*proof*⟩

**lemma** *rADnMT*[*rule-format*]: $p \neq []$ ⟶ *removeAllDuplicates p* $\neq []$
⟨*proof*⟩

**lemma** *remDupsNMT*[*rule-format*]: $p \neq []$ ⟶
*remdups p* $\neq []$
⟨*proof*⟩

**lemma** *sets-distinct1*: (*n*::*int*) $\neq m \Longrightarrow \{(a,b).\ a = n\} \neq \{(a,b).\ a = m\}$
⟨*proof*⟩

**lemma** *sets-distinct2*: (*m*::*int*) $\neq n \Longrightarrow \{(a,b).\ a = n\} \neq \{(a,b).\ a = m\}$
⟨*proof*⟩

**lemma** *sets-distinct5*: (*n*::*int*) $< m \Longrightarrow \{(a,b).\ a = n\} \neq \{(a,b).\ a = m\}$
⟨*proof*⟩

**lemma** *sets-distinct6*: (*m*::*int*) $< n \Longrightarrow \{(a,b).\ a = n\} \neq \{(a,b).\ a = m\}$
⟨*proof*⟩

**end**

**theory** *NormalisationIntegerPortProof*
**imports** *NormalisationGenericProofs*
**begin**

Normalisation proofs which are specific to the IntegerPort address representation.

**lemma** *ConcAssoc*: $C((A \oplus B) \oplus D) = C(A \oplus (B \oplus D))$
⟨*proof*⟩

**lemma** *aux26*[*simp*]: *twoNetsDistinct a b c d* $\Longrightarrow$
*dom* (*C* (*AllowPortFromTo a b p*)) $\cap$ *dom* (*C* (*DenyAllFromTo c d*)) = {}
⟨*proof*⟩

**lemma** *wp2-aux*[*rule-format*]: *wellformed-policy2* (*xs* @ [*x*]) $\longrightarrow$
*wellformed-policy2 xs*
⟨*proof*⟩

**lemma** *Cdom2*: *x* $\in$ *dom*(*C b*) $\Longrightarrow$ *C* (*a* $\oplus$ *b*) *x* = (*C b*) *x*
⟨*proof*⟩

**lemma** *wp2Conc*[*rule-format*]: *wellformed-policy2* (*x*#*xs*) $\Longrightarrow$ *wellformed-policy2 xs*
⟨*proof*⟩

**lemma** *DAimpliesMR-E*[*rule-format*]: *DenyAll* $\in$ *set p* $\longrightarrow$
(∃ *r*. *applied-rule-rev C x p* = *Some r*)
⟨*proof*⟩

**lemma** *DAimplieMR*[*rule-format*]: *DenyAll* $\in$ *set p* $\Longrightarrow$ *applied-rule-rev C x p* $\neq$ *None*
⟨*proof*⟩

**lemma** *MRList1*[*rule-format*]: *x* $\in$ *dom* (*C a*) $\Longrightarrow$ *applied-rule-rev C x* (*b*@[*a*]) = *Some a*
⟨*proof*⟩

**lemma** *MRList2*: *x* $\in$ *dom* (*C a*) $\Longrightarrow$ *applied-rule-rev C x* (*c*@*b*@[*a*]) = *Some a*
⟨*proof*⟩

**lemma** *MRList3*: *x* $\notin$ *dom* (*C xa*) $\Longrightarrow$
*applied-rule-rev C x* (*a* @ *b* # *xs* @ [*xa*]) = *applied-rule-rev C x* (*a* @ *b* # *xs*)
⟨*proof*⟩

**lemma** *CConcEnd*[*rule-format*]: *C a x* = *Some y* $\longrightarrow$
*C* (*list2FWpolicy* (*xs* @ [*a*])) *x* = *Some y*
(**is** *?P xs*)
⟨*proof*⟩

**lemma** *CConcStartaux*: ⟦*C a x* = *None*⟧ $\Longrightarrow$ (*C aa* ++ *C a*) *x* = *C aa x*
⟨*proof*⟩

**lemma** *CConcStart*[*rule-format*]: *xs* $\neq$ [] $\longrightarrow$ *C a x* = *None* $\longrightarrow$
*C* (*list2FWpolicy* (*xs* @ [*a*])) *x* = *C* (*list2FWpolicy xs*) *x*
⟨*proof*⟩

**lemma** *mrNnt*[*simp*]: *applied-rule-rev C x p* = *Some a* $\Longrightarrow$ *p* $\neq$ []
⟨*proof*⟩

**lemma** *mr-is-C*[*rule-format*]: *applied-rule-rev C x p = Some a ⟶*
*C (list2FWpolicy (p)) x = C a x*
⟨*proof*⟩

**lemma** *CConcStart2*: ⟦*p ≠ []*; *x ∉ dom (C a)*⟧ *⟹*
*C (list2FWpolicy (p@[a])) x = C (list2FWpolicy p)x*
⟨*proof*⟩

**lemma** *CConcEnd1*: ⟦*q@p ≠ []*; *x ∉ dom (C a)*⟧ *⟹*
*C (list2FWpolicy (q@p@[a])) x = C (list2FWpolicy (q@p))x*
⟨*proof*⟩

**lemma** *CConcEnd2*[*rule-format*]: *x ∈ dom (C a) ⟶*
*C (list2FWpolicy (xs @ [a])) x = C a x*
(**is** *?P xs*)
⟨*proof*⟩

**lemma** *bar3*: *x ∈ dom (C (list2FWpolicy (xs @ [xa]))) ⟹*
*x ∈ dom (C (list2FWpolicy xs)) ∨ x ∈ dom (C xa)*
⟨*proof*⟩

**lemma** *CeqEnd*[*rule-format*,*simp*]: *a ≠ [] ⟶ x ∈ dom (C (list2FWpolicy a)) ⟶*
*C (list2FWpolicy (b@a)) x = (C (list2FWpolicy a)) x*
⟨*proof*⟩

**lemma** *CConcStartA*[*rule-format*,*simp*]: *x ∈ dom (C a) ⟶*
*x ∈ dom (C (list2FWpolicy (a # b)))*
(**is** *?P b*)
⟨*proof*⟩

**lemma** *domConc*: ⟦*x ∈ dom (C (list2FWpolicy b))*; *b ≠ []*⟧ *⟹*
*x ∈ dom (C (list2FWpolicy (a@b)))*
⟨*proof*⟩

**lemma** *CeqStart*[*rule-format*,*simp*]:
*x ∉ dom (C (list2FWpolicy a)) ⟶ a ≠ [] ⟶ b ≠ [] ⟶*
*C (list2FWpolicy (b@a)) x = (C (list2FWpolicy b)) x*
⟨*proof*⟩

**lemma** *C-eq-if-mr-eq2*: ⟦*applied-rule-rev C x a = Some r*; *applied-rule-rev C x b = Some r*;
*a ≠ []*; *b ≠ []*⟧ *⟹*
*(C (list2FWpolicy a)) x = (C (list2FWpolicy b)) x*
⟨*proof*⟩

**lemma** *nMRtoNone*[*rule-format*]: *p ≠ [] ⟶ applied-rule-rev C x p = None ⟶*

$$C\ (list2FWpolicy\ p)\ x = None$$

⟨*proof*⟩

**lemma** *C-eq-if-mr-eq*:
⟦*applied-rule-rev C x b* = *applied-rule-rev C x a*; *a* ≠ []; *b* ≠ []⟧ ⟹
(*C* (*list2FWpolicy a*)) *x* = (*C* (*list2FWpolicy b*)) *x*
⟨*proof*⟩

**lemma** *notmatching-notdom*: *applied-rule-rev C x* (*p*@[*a*]) ≠ *Some a* ⟹ *x* ∉ *dom* (*C a*)
⟨*proof*⟩

**lemma** *foo3a*[*rule-format*]: *applied-rule-rev C x* (*a*@[*b*]@*c*) = *Some b* ⟶ *r* ∈ *set c* ⟶
*b* ∉ *set c* ⟶ *x* ∉ *dom* (*C r*)
⟨*proof*⟩

**lemma** *foo3D*: ⟦*wellformed-policy1 p*; *p* = (*DenyAll*#*ps*);
*applied-rule-rev C x p* = *Some DenyAll*; *r* ∈ *set ps*⟧ ⟹ *x* ∉ *dom* (*C r*)
⟨*proof*⟩

**lemma** *foo4*[*rule-format*]: *set p* = *set s* ∧ (∀ *r*. *r* ∈ *set p* ⟶ *x* ∉ *dom* (*C r*)) ⟶
(∀ *r* .*r* ∈ *set s* ⟶ *x* ∉ *dom* (*C r*))
⟨*proof*⟩

**lemma** *foo5b*[*rule-format*]: *x* ∈ *dom* (*C b*) ⟶ (∀ *r*. *r* ∈ *set c* ⟶ *x* ∉ *dom* (*C r*))⟶
*applied-rule-rev C x* (*b*#*c*) = *Some b*
⟨*proof*⟩

**lemma** *mr-first*: ⟦*x* ∈ *dom* (*C b*); (∀ *r*. *r* ∈ *set c* ⟶ *x* ∉ *dom* (*C r*)); *s* = *b*#*c*⟧ ⟹
*applied-rule-rev C x s* = *Some b*
⟨*proof*⟩

**lemma** *mr-charn*[*rule-format*]: *a* ∈ *set p* ⟶ (*x* ∈ *dom* (*C a*)) ⟶
(∀ *r*. *r* ∈ *set p* ∧ *x* ∈ *dom* (*C r*) ⟶ *r* = *a*) ⟶
*applied-rule-rev C x p* = *Some a*
⟨*proof*⟩

**lemma** *foo8*: ⟦(∀ *r*. *r* ∈ *set p* ∧ *x* ∈ *dom* (*C r*) ⟶ *r* = *a*); *set p* = *set s*⟧ ⟹
(∀ *r*. *r* ∈ *set s* ∧ *x* ∈ *dom* (*C r*) ⟶ *r* = *a*)
⟨*proof*⟩

**lemma** *mrConcEnd*[*rule-format*]: *applied-rule-rev C x* (*b* # *p*) = *Some a* ⟶ *a* ≠ *b* ⟶
*applied-rule-rev C x p* = *Some a*
⟨*proof*⟩

**lemma** *wp3tl*[*rule-format*]: *wellformed-policy3 p* ⟶ *wellformed-policy3* (*tl p*)
⟨*proof*⟩

**lemma** *wp3Conc*[*rule-format*]: *wellformed-policy3* $(a\#p) \longrightarrow$ *wellformed-policy3* $p$
⟨*proof*⟩

**lemma** *foo98*[*rule-format*]:*applied-rule-rev* $C\ x\ (aa\ \#\ p) = Some\ a \longrightarrow x \in dom\ (C\ r) \longrightarrow$
$r \in set\ p$
$\longrightarrow a \in set\ p$
⟨*proof*⟩

**lemma** *mrMTNone*[*simp*]: *applied-rule-rev* $C\ x\ [] = None$
⟨*proof*⟩

**lemma** *DAAux*[*simp*]: $x \in dom\ (C\ DenyAll)$
⟨*proof*⟩

**lemma** *mrSet*[*rule-format*]: *applied-rule-rev* $C\ x\ p = Some\ r \longrightarrow r \in set\ p$
⟨*proof*⟩

**lemma** *mr-not-Conc*: *singleCombinators* $p \Longrightarrow$ *applied-rule-rev* $C\ x\ p \neq Some\ (a \oplus b)$
⟨*proof*⟩

**lemma** *foo25*[*rule-format*]: *wellformed-policy3* $(p@[x]) \longrightarrow$ *wellformed-policy3* $p$
⟨*proof*⟩

**lemma** *mr-in-dom*[*rule-format*]: *applied-rule-rev* $C\ x\ p = Some\ a \longrightarrow x \in dom\ (C\ a)$
⟨*proof*⟩

**lemma** *wp3EndMT*[*rule-format*]: *wellformed-policy3* $(p@[xs]) \longrightarrow$
*AllowPortFromTo* $a\ b\ po \in set\ p \longrightarrow$
$dom\ (C\ (AllowPortFromTo\ a\ b\ po)) \cap dom\ (C\ xs) = \{\}$
⟨*proof*⟩

**lemma** *foo29*: $[\![dom\ (C\ a) \neq \{\};\ dom\ (C\ a) \cap dom\ (C\ b) = \{\}]\!] \Longrightarrow a \neq b$
⟨*proof*⟩

**lemma** *foo28*: $[\![AllowPortFromTo\ a\ b\ po \in set\ p;$
$dom\ (C\ (AllowPortFromTo\ a\ b\ po)) \neq \{\};$ (*wellformed-policy3* $(p@[x]))]\!]$
$\Longrightarrow x \neq AllowPortFromTo\ a\ b\ po$
⟨*proof*⟩

**lemma** *foo28a*[*rule-format*]: $x \in dom\ (C\ a) \Longrightarrow dom\ (C\ a) \neq \{\}$
⟨*proof*⟩

**lemma** *allow-deny-dom*[*simp*]: $dom\ (C\ (AllowPortFromTo\ a\ b\ po)) \subseteq$
$dom\ (C\ (DenyAllFromTo\ a\ b))$

⟨*proof*⟩

**lemma** *DenyAllowDisj*: *dom* (*C* (*AllowPortFromTo a b p*)) ≠ {} ⟹
*dom* (*C* (*DenyAllFromTo a b*)) ∩ *dom* (*C* (*AllowPortFromTo a b p*)) ≠ {}
⟨*proof*⟩

**lemma** *foo31*: ⟦(∀ *r*. *r* ∈ *set p* ∧ *x* ∈ *dom* (*C r*) ⟶
(*r* = *AllowPortFromTo a b po* ∨ *r* = *DenyAllFromTo a b* ∨ *r* = *DenyAll*));
*set p* = *set s*⟧ ⟹
(∀ *r*. *r* ∈ *set s* ∧ *x* ∈ *dom* (*C r*) ⟶
(*r* = *AllowPortFromTo a b po* ∨ *r* = *DenyAllFromTo a b* ∨ *r* = *DenyAll*))
⟨*proof*⟩

**lemma** *wp1-auxa*: *wellformed-policy1-strong p*⟹(∃ *r*. *applied-rule-rev C x p* = *Some r*)
⟨*proof*⟩

**lemma** *deny-dom*[*simp*]: *twoNetsDistinct a b c d* ⟹ *dom* (*C* (*DenyAllFromTo a b*)) ∩
*dom* (*C* (*DenyAllFromTo c d*)) = {}
⟨*proof*⟩

**lemma** *domTrans*: ⟦*dom a* ⊆ *dom b*; *dom*(*b*) ∩ *dom* (*c*) = {}⟧ ⟹ *dom*(*a*) ∩ *dom*(*c*) = {}
⟨*proof*⟩

**lemma** *DomInterAllowsMT*: ⟦ *twoNetsDistinct a b c d*⟧ ⟹
*dom* (*C* (*AllowPortFromTo a b p*)) ∩ *dom* (*C* (*AllowPortFromTo c d po*)) = {}
⟨*proof*⟩

**lemma** *DomInterAllowsMT-Ports*: ⟦*p* ≠ *po*⟧ ⟹
*dom* (*C* (*AllowPortFromTo a b p*)) ∩ *dom* (*C* (*AllowPortFromTo c d po*)) = {}
⟨*proof*⟩

**lemma** *wellformed-policy3-charn*[*rule-format*]:
*singleCombinators p* ⟶ *distinct p* ⟶ *allNetsDistinct p* ⟶
*wellformed-policy1 p* ⟶ *wellformed-policy2 p* ⟶ *wellformed-policy3 p*
⟨*proof*⟩

**lemma** *DistinctNetsDenyAllow*:
⟦*DenyAllFromTo b c* ∈ *set p*; *AllowPortFromTo a d po* ∈ *set p*; *allNetsDistinct p*;
*dom* (*C* (*DenyAllFromTo b c*)) ∩ *dom* (*C* (*AllowPortFromTo a d po*)) ≠ {}⟧
⟹ *b* = *a* ∧ *c* = *d*
⟨*proof*⟩

**lemma** *DistinctNetsAllowAllow*:
⟦*AllowPortFromTo b c poo* ∈ *set p*; *AllowPortFromTo a d po* ∈ *set p*;
*allNetsDistinct p*; *dom* (*C* (*AllowPortFromTo b c poo*)) ∩
*dom* (*C* (*AllowPortFromTo a d po*)) ≠ {}⟧
⟹ *b* = *a* ∧ *c* = *d* ∧ *poo* = *po*
⟨*proof*⟩

**lemma** *WP2RS2*[*simp*]:
⟦*singleCombinators p*;
*distinct p*;
*allNetsDistinct p*⟧ ⟹
*wellformed-policy2* (*removeShadowRules2 p*)
⟨*proof*⟩

**lemma** *AD-aux*: ⟦*AllowPortFromTo a b po* ∈ *set p* ;*DenyAllFromTo c d* ∈ *set p*;
*allNetsDistinct p* ; *singleCombinators p*;
*a* ≠ *c* ∨ *b* ≠ *d*⟧
⟹ *dom* (*C* (*AllowPortFromTo a b po*)) ∩ *dom* (*C* (*DenyAllFromTo c d*)) = {}
⟨*proof*⟩

**lemma** *sorted-WP2*[*rule-format*]: *sorted p l* ⟶ *all-in-list p l* ⟶ *distinct p* ⟶
*allNetsDistinct p* ⟶ *singleCombinators p* ⟶ *wellformed-policy2 p*
⟨*proof*⟩

**lemma** *wellformed2-sorted*[*simp*]: ⟦*all-in-list p l*; *distinct p*; *allNetsDistinct p*;
*singleCombinators p*⟧ ⟹ *wellformed-policy2* (*sort p l*)
⟨*proof*⟩

**lemma** *wellformed2-sortedQ*[*simp*]: ⟦*all-in-list p l*; *distinct p*; *allNetsDistinct p*;
*singleCombinators p*⟧ ⟹ *wellformed-policy2* (*qsort p l*)
⟨*proof*⟩

**lemma** *C-DenyAll*[*simp*]: *C* (*list2FWpolicy* (*xs* @ [*DenyAll*])) *x* = *Some* (*deny* ())
⟨*proof*⟩

**lemma** *C-eq-RS1n*:

$C(\mathit{list2FWpolicy}\ (\mathit{removeShadowRules1\text{-}alternative}\ p)) = C(\mathit{list2FWpolicy}\ p)$
⟨*proof*⟩

**lemma** *C-eq-RS1*[*simp*]: $p \neq [] \Longrightarrow$
$C(\mathit{list2FWpolicy}\ (\mathit{removeShadowRules1}\ p)) = C(\mathit{list2FWpolicy}\ p)$
⟨*proof*⟩

**lemma** *EX-MR-aux*[*rule-format*]: $\mathit{applied\text{-}rule\text{-}rev}\ C\ x\ (\mathit{DenyAll}\ \#\ p) \neq \mathit{Some}\ \mathit{DenyAll} \longrightarrow$
$(\exists\, y.\ \mathit{applied\text{-}rule\text{-}rev}\ C\ x\ p = \mathit{Some}\ y)$
⟨*proof*⟩

**lemma** *EX-MR* : $[\![\mathit{applied\text{-}rule\text{-}rev}\ C\ x\ p \neq (\mathit{Some}\ \mathit{DenyAll});\ p = \mathit{DenyAll}\#ps]\!] \Longrightarrow$
$(\mathit{applied\text{-}rule\text{-}rev}\ C\ x\ p = \mathit{applied\text{-}rule\text{-}rev}\ C\ x\ ps)$
⟨*proof*⟩

**lemma** *mr-not-DA*:
$[\![\mathit{wellformed\text{-}policy1\text{-}strong}\ s;\ \mathit{applied\text{-}rule\text{-}rev}\ C\ x\ p = \mathit{Some}\ (\mathit{DenyAllFromTo}\ a\ ab);$
$\mathit{set}\ p = \mathit{set}\ s]\!] \Longrightarrow \mathit{applied\text{-}rule\text{-}rev}\ C\ x\ s \neq \mathit{Some}\ \mathit{DenyAll}$
⟨*proof*⟩

**lemma** *domsMT-notND-DD*:
$[\![\mathit{dom}\ (C\ (\mathit{DenyAllFromTo}\ a\ b)) \cap \mathit{dom}\ (C\ (\mathit{DenyAllFromTo}\ c\ d)) \neq \{\}]\!] \Longrightarrow$
$\neg\ \mathit{netsDistinct}\ a\ c$
⟨*proof*⟩

**lemma** *domsMT-notND-DD2*:
$[\![\mathit{dom}\ (C\ (\mathit{DenyAllFromTo}\ a\ b)) \cap \mathit{dom}\ (C\ (\mathit{DenyAllFromTo}\ c\ d)) \neq \{\}]\!] \Longrightarrow$
$\neg\ \mathit{netsDistinct}\ b\ d$
⟨*proof*⟩

**lemma** *domsMT-notND-DD3*:
$[\![x \in \mathit{dom}\ (C\ (\mathit{DenyAllFromTo}\ a\ b));\ x \in \mathit{dom}\ (C\ (\mathit{DenyAllFromTo}\ c\ d))]\!] \Longrightarrow$
$\neg\ \mathit{netsDistinct}\ a\ c$
⟨*proof*⟩

**lemma** *domsMT-notND-DD4*:
$[\![x \in \mathit{dom}\ (C\ (\mathit{DenyAllFromTo}\ a\ b));\ x \in \mathit{dom}\ (C\ (\mathit{DenyAllFromTo}\ c\ d))]\!] \Longrightarrow$
$\neg\ \mathit{netsDistinct}\ b\ d$
⟨*proof*⟩

**lemma** *NetsEq-if-sameP-DD*:
$[\![\mathit{allNetsDistinct}\ p;\ u \in \mathit{set}\ p;\ v \in \mathit{set}\ p;\ u = (\mathit{DenyAllFromTo}\ a\ b);$
$v = (\mathit{DenyAllFromTo}\ c\ d);\ x \in \mathit{dom}\ (C\ (u));\ x \in \mathit{dom}\ (C\ (v))]\!] \Longrightarrow$
$a = c \wedge b = d$

⟨*proof*⟩

**lemma** *rule-charn1*:
**assumes** *aND*: *allNetsDistinct p*
**and** *mr-is-allow*: *applied-rule-rev C x p* = *Some* (*AllowPortFromTo a b po*)
**and** *SC*: *singleCombinators p*
**and** *inp*: $r \in set\ p$
**and** *inDom*: $x \in dom\ (C\ r)$
**shows** ($r$ = *AllowPortFromTo a b po* ∨ $r$ = *DenyAllFromTo a b* ∨ $r$ = *DenyAll*)
⟨*proof*⟩

**lemma** *noneMTsubset*[*rule-format*]: *noneMT C a* ⟶ *set b* ⊆ *set a* ⟶ *noneMT C b*
⟨*proof*⟩

**lemma** *nMTSort*: *noneMT C p* ⟹ *noneMT C* (*sort p l*)
⟨*proof*⟩

**lemma** *nMTSortQ*: *noneMT C p* ⟹ *noneMT C* (*qsort p l*)
⟨*proof*⟩

**lemma** *wp3char*[*rule-format*]: *noneMT C xs* ∧ *C* (*AllowPortFromTo a b po*) = *empty* ∧
*wellformed-policy3* (*xs* @ [*DenyAllFromTo a b*]) ⟶
*AllowPortFromTo a b po* ∉ *set xs*
⟨*proof*⟩

**lemma** *wp3charn*[*rule-format*]:
**assumes** *domAllow*: *dom* (*C* (*AllowPortFromTo a b po*)) ≠ {}
**and** *wp3*: *wellformed-policy3* (*xs* @ [*DenyAllFromTo a b*])
**shows** *allowNotInList*: *AllowPortFromTo a b po* ∉ *set xs*
⟨*proof*⟩

**lemma** *rule-charn2*:
**assumes** *aND*: *allNetsDistinct p*
**and** *wp1*: *wellformed-policy1 p*
**and** *SC*: *singleCombinators p*
**and** *wp3*: *wellformed-policy3 p*
**and** *allow-in-list*: *AllowPortFromTo c d po* ∈ *set p*
**and** *x-in-dom-allow*: $x \in dom\ (C\ (AllowPortFromTo\ c\ d\ po))$
**shows** *applied-rule-rev C x p* = *Some* (*AllowPortFromTo c d po*)

⟨*proof*⟩

**lemma** *rule-charn3*:
⟦*wellformed-policy1 p*; *allNetsDistinct p*; *singleCombinators p*;
*wellformed-policy3 p*; *applied-rule-rev C x p* = *Some* (*DenyAllFromTo c d*);
*AllowPortFromTo a b po* ∈ *set p*⟧ ⟹ *x* ∉ *dom* (*C* (*AllowPortFromTo a b po*))
⟨*proof*⟩

**lemma** *rule-charn4*:
**assumes** *wp1*: *wellformed-policy1 p*
**and** *aND*: *allNetsDistinct p*
**and** *SC*: *singleCombinators p*
**and** *wp3*: *wellformed-policy3 p*
**and** *DA*: *DenyAll* ∉ *set p*
**and** *mr*: *applied-rule-rev C x p* = *Some* (*DenyAllFromTo a b*)
**and** *rinp*: *r* ∈ *set p*
**and** *xindom*: *x* ∈ *dom* (*C r*)
**shows** *r* = *DenyAllFromTo a b*
⟨*proof*⟩

**lemma** *foo31a*: ⟦(∀ *r*. *r* ∈ *set p* ∧ *x* ∈ *dom* (*C r*) ⟶
(*r* = *AllowPortFromTo a b po* ∨ *r* = *DenyAllFromTo a b* ∨ *r* = *DenyAll*));
*set p* = *set s*; *r* ∈ *set s* ; *x* ∈ *dom* (*C r*)⟧ ⟹
(*r* = *AllowPortFromTo a b po* ∨ *r* = *DenyAllFromTo a b* ∨ *r* = *DenyAll*)
⟨*proof*⟩

**lemma** *aux4*[*rule-format*]:
*applied-rule-rev C x* (*a*#*p*) = *Some a* ⟶ *a* ∉ *set* (*p*) ⟶ *applied-rule-rev C x p* = *None*
⟨*proof*⟩

**lemma** *mrDA-tl*:
**assumes** *mr-DA*: *applied-rule-rev C x p* = *Some DenyAll*
**and** *wp1n*: *wellformed-policy1-strong p*
**shows** *applied-rule-rev C x* (*tl p*) = *None*
⟨*proof*⟩

**lemma** *rule-charnDAFT*:
⟦*wellformed-policy1-strong p*; *allNetsDistinct p*; *singleCombinators p*;
*wellformed-policy3 p*; *applied-rule-rev C x p* = *Some* (*DenyAllFromTo a b*);
*r* ∈ *set* (*tl p*); *x* ∈ *dom* (*C r*)⟧ ⟹ *r* = *DenyAllFromTo a b*
⟨*proof*⟩

**lemma** *mrDenyAll-is-unique*:
⟦*wellformed-policy1-strong p*; *applied-rule-rev C x p* = *Some DenyAll*;
*r* ∈ *set* (*tl p*)⟧ ⟹ *x* ∉ *dom* (*C r*)
⟨*proof*⟩

**theorem** *C-eq-Sets-mr*:
**assumes** *sets-eq*: *set p = set s*
**and** *SC*: *singleCombinators p*
**and** *wp1-p*: *wellformed-policy1-strong p*
**and** *wp1-s*: *wellformed-policy1-strong s*
**and** *wp3-p*: *wellformed-policy3 p*
**and** *wp3-s*: *wellformed-policy3 s*
**and** *aND*: *allNetsDistinct p*

**shows** *applied-rule-rev C x p = applied-rule-rev C x s*
⟨*proof*⟩

**lemma** *C-eq-Sets*:
⟦*singleCombinators p*; *wellformed-policy1-strong p*; *wellformed-policy1-strong s*;
*wellformed-policy3 p*; *wellformed-policy3 s*; *allNetsDistinct p*; *set p = set s*⟧ ⟹
*C* (*list2FWpolicy p*) *x* = *C* (*list2FWpolicy s*) *x*
⟨*proof*⟩

**lemma** *C-eq-sorted*: ⟦*distinct p*; *all-in-list p l*; *singleCombinators p*;
*wellformed-policy1-strong p*; *wellformed-policy3 p*; *allNetsDistinct p*⟧ ⟹
*C* (*list2FWpolicy* (*sort p l*))= *C* (*list2FWpolicy p*)
⟨*proof*⟩

**lemma** *C-eq-sortedQ*: ⟦*distinct p*; *all-in-list p l*; *singleCombinators p*;
*wellformed-policy1-strong p*; *wellformed-policy3 p*; *allNetsDistinct p*⟧ ⟹
*C* (*list2FWpolicy* (*qsort p l*))= *C* (*list2FWpolicy p*)
⟨*proof*⟩

**lemma** *C-eq-RS2-mr*: *applied-rule-rev C x* (*removeShadowRules2 p*)= *applied-rule-rev C x p*
⟨*proof*⟩

**lemma** *C-eq-None*[*rule-format*]: *p* ≠ [] −−> *applied-rule-rev C x p = None* ⟶
*C* (*list2FWpolicy p*) *x = None*
⟨*proof*⟩

**lemma** *C-eq-None2*:
⟦*a* ≠ []; *b* ≠ []; *applied-rule-rev C x a = None*; *applied-rule-rev C x b = None*⟧ ⟹
(*C* (*list2FWpolicy a*)) *x* = (*C* (*list2FWpolicy b*)) *x*
⟨*proof*⟩

**lemma** *C-eq-RS2*: *wellformed-policy1-strong p* $\Longrightarrow$
*C* (*list2FWpolicy* (*removeShadowRules2 p*))= *C* (*list2FWpolicy p*)
⟨*proof*⟩

**lemma** *noneMTRS2*: *noneMT C p* $\Longrightarrow$ *noneMT C* (*removeShadowRules2 p*)
⟨*proof*⟩

**lemma** *CconcNone*: ⟦*dom* (*C a*) = {}; *p* ≠ []⟧ $\Longrightarrow$
*C* (*list2FWpolicy* (*a* # *p*)) *x* = *C* (*list2FWpolicy p*) *x*
⟨*proof*⟩

**lemma** *noneMTrd*[*rule-format*]: *noneMT C p* $\longrightarrow$ *noneMT C* (*remdups p*)
⟨*proof*⟩

**lemma** *DARS3*[*rule-format*]:*DenyAll* ∉ *set p*$\longrightarrow$*DenyAll* ∉ *set* (*removeShadowRules3 C p*)
⟨*proof*⟩

**lemma** *DAnMT*: *dom* (*C DenyAll*) ≠ {}
⟨*proof*⟩

**lemma** *DAnMT2*: *C DenyAll* ≠ *empty*
⟨*proof*⟩

**lemma** *wp1n-RS3*[*rule-format*,*simp*]: *wellformed-policy1-strong p* $\longrightarrow$
*wellformed-policy1-strong* (*removeShadowRules3 C p*)
⟨*proof*⟩

**lemma** *AILRS3*[*rule-format*,*simp*]: *all-in-list p l* $\longrightarrow$
*all-in-list* (*removeShadowRules3 C p*) *l*
⟨*proof*⟩

**lemma** *SCRS3*[*rule-format*,*simp*]: *singleCombinators p* $\longrightarrow$
*singleCombinators*(*removeShadowRules3 C p*)
⟨*proof*⟩

**lemma** *RS3subset*: *set* (*removeShadowRules3 C p*) ⊆ *set p*
⟨*proof*⟩

**lemma** *ANDRS3*[*simp*]: ⟦*singleCombinators p*; *allNetsDistinct p*⟧ $\Longrightarrow$
*allNetsDistinct* (*removeShadowRules3 C p*)

⟨*proof*⟩

**lemma** *nlpaux*: $x \notin dom\ (C\ b) \Longrightarrow C\ (a \oplus b)\ x = C\ a\ x$
⟨*proof*⟩

**lemma** *notindom*[*rule-format*]: $a \in set\ p \longrightarrow\ x \notin dom\ (C\ (list2FWpolicy\ p)) \longrightarrow$
$x \notin dom\ (C\ a)$
⟨*proof*⟩

**lemma** *C-eq-rd*[*rule-format*]: $p \neq [] \Longrightarrow$
$C\ (list2FWpolicy\ (remdups\ p)) = C\ (list2FWpolicy\ p)$
⟨*proof*⟩

**lemma** *nMT-domMT*: $[\![\neg\ notMTpolicy\ C\ \ p;\ p \neq []]\!] \Longrightarrow r \notin dom\ (C\ (list2FWpolicy\ p))$
⟨*proof*⟩

**lemma** *C-eq-RS3-aux*[*rule-format*]: $notMTpolicy\ C\ \ p \Longrightarrow$
$C\ (list2FWpolicy\ p)\ x = C\ (list2FWpolicy\ (removeShadowRules3\ C\ p))\ x$
⟨*proof*⟩

**lemma** *C-eq-id*: $wellformed\text{-}policy1\text{-}strong\ p \Longrightarrow$
$C(list2FWpolicy\ (insertDeny\ p)) = C\ (list2FWpolicy\ p)$
⟨*proof*⟩

**lemma** *C-eq-RS3*: $notMTpolicy\ C\ \ p \Longrightarrow$
$C(list2FWpolicy\ (removeShadowRules3\ C\ p)) = C\ (list2FWpolicy\ p)$
⟨*proof*⟩

**lemma** *NMPrd*[*rule-format*]: $notMTpolicy\ C\ \ p \longrightarrow notMTpolicy\ C\ \ (remdups\ p)$
⟨*proof*⟩

**lemma** *NMPDA*[*rule-format*]: $DenyAll \in set\ p \longrightarrow notMTpolicy\ C\ \ p$
⟨*proof*⟩

**lemma** *NMPiD*[*rule-format*]: $notMTpolicy\ C\ \ (insertDeny\ p)$
⟨*proof*⟩

**lemma** *list2FWpolicy2list*[*rule-format*]: $C\ (list2FWpolicy(policy2list\ p)) = (C\ p)$

⟨*proof*⟩

**lemmas** *C-eq-Lemmas = noneMTRS2 noneMTrd SCp2l*
*wp1n-RS2 wp1ID NMPiD wp1alternative-RS1*
*p2lNmt list2FWpolicy2list wellformed-policy3-charn waux2 wp1-eq*

**lemmas** *C-eq-subst-Lemmas = C-eq-sorted C-eq-sortedQ C-eq-RS2 C-eq-rd C-eq-RS3 C-eq-id*

**lemma** *C-eq-All-untilSorted*:
⟦*DenyAll ∈ set (policy2list p); all-in-list (policy2list p) l;*
*allNetsDistinct (policy2list p)*⟧ ⟹
*C*(*list2FWpolicy (sort (removeShadowRules2 (remdups (removeShadowRules3 C*
*(insertDeny (removeShadowRules1 (policy2list p)))))) l)) = C p*
⟨*proof*⟩

**lemma** *C-eq-All-untilSortedQ*:
⟦*DenyAll ∈ set (policy2list p); all-in-list (policy2list p) l;*
*allNetsDistinct (policy2list p)*⟧ ⟹
*C*(*list2FWpolicy (qsort (removeShadowRules2 (remdups (removeShadowRules3 C*
*(insertDeny (removeShadowRules1 (policy2list p)))))) l)) = C p*
⟨*proof*⟩

**lemma** *C-eq-All-untilSorted-withSimps*:
⟦*DenyAll ∈ set (policy2list p); all-in-list (policy2list p) l;*
*allNetsDistinct (policy2list p)*⟧ ⟹
*C*(*list2FWpolicy (sort (removeShadowRules2 (remdups (removeShadowRules3 C (insertDeny*
*(removeShadowRules1 (policy2list p)))))) l)) = C p*
⟨*proof*⟩

**lemma** *C-eq-All-untilSorted-withSimpsQ*:
⟦*DenyAll ∈ set (policy2list p); all-in-list (policy2list p) l;*
*allNetsDistinct (policy2list p)*⟧ ⟹
*C*(*list2FWpolicy (qsort (removeShadowRules2 (remdups (removeShadowRules3 C (insertDeny*
*(removeShadowRules1 (policy2list p)))))) l)) = C p*
⟨*proof*⟩

**lemma** *InDomConc*[*rule-format*]: *p ≠ [] ⟶ x ∈ dom (C (list2FWpolicy (p))) ⟶*
*x ∈ dom (C (list2FWpolicy (a#p)))*
⟨*proof*⟩

**lemma** *not-in-member*[*rule-format*]: *member a b* $\longrightarrow$ $x \notin$ *dom* (*C b*) $\longrightarrow$ $x \notin$ *dom* (*C a*)
⟨*proof*⟩

**lemma** *src-in-sdnets*[*rule-format*]: ¬ *member DenyAll x* $\longrightarrow$ $p \in$ *dom* (*C x*) $\longrightarrow$
*subnetsOfAdr* (*src p*) ∩ (*fst-set* (*sdnets x*)) ≠ {}
⟨*proof*⟩

**lemma** *dest-in-sdnets*[*rule-format*]: ¬ *member DenyAll x* $\longrightarrow$ $p \in$ *dom* (*C x*) $\longrightarrow$
*subnetsOfAdr* (*dest p*) ∩ (*snd-set* (*sdnets x*)) ≠ {}
⟨*proof*⟩

**lemma** *sdnets-in-subnets*[*rule-format*]: *p*∈ *dom* (*C x*) $\longrightarrow$ ¬ *member DenyAll x* $\longrightarrow$
(∃ (*a*,*b*)∈*sdnets x*. *a* ∈ *subnetsOfAdr* (*src p*) ∧ *b* ∈ *subnetsOfAdr* (*dest p*))
⟨*proof*⟩

**lemma** *disjSD-no-p-in-both*[*rule-format*]:
⟦*disjSD-2 x y*; ¬ *member DenyAll x*; ¬ *member DenyAll y*;
*p* ∈ *dom* (*C x*); *p* ∈ *dom* (*C y*)⟧ ⟹ *False*
⟨*proof*⟩

**lemma** *list2FWpolicy-eq*: *zs* ≠ [] ⟹
*C* (*list2FWpolicy* (*x* ⊕ *y* # *z*)) *p* = *C* (*x* ⊕ *list2FWpolicy* (*y* # *z*)) *p*
⟨*proof*⟩

**lemma** *dom-sep*[*rule-format*]: *x* ∈ *dom* (*C* (*list2FWpolicy p*)) $\longrightarrow$
*x* ∈ *dom* (*C* (*list2FWpolicy*(*separate p*)))
⟨*proof*⟩

**lemma** *domdConcStart*[*rule-format*]: *x* ∈ *dom* (*C* (*list2FWpolicy* (*a*#*b*))) $\longrightarrow$
*x* ∉ *dom* (*C* (*list2FWpolicy b*))
$\longrightarrow$ *x* ∈ *dom* (*C* (*a*))
⟨*proof*⟩

**lemma** *sep-dom2-aux*: ⟦*x* ∈ *dom* (*C* (*list2FWpolicy* (*a* ⊕ *y* # *z*)))⟧
⟹ *x* ∈ *dom* (*C* (*a* ⊕ *list2FWpolicy* (*y* # *z*)))
⟨*proof*⟩

**lemma** *sep-dom2-aux2*:
⟦(*x* ∈ *dom* (*C* (*list2FWpolicy* (*separate* (*y* # *z*)))) $\longrightarrow$
*x* ∈ *dom* (*C* (*list2FWpolicy* (*y* # *z*))));
*x* ∈ *dom* (*C* (*list2FWpolicy* (*a* # *separate* (*y* # *z*))))⟧

$\Longrightarrow x \in dom\ (C\ (list2FWpolicy\ (a \oplus y\ \#\ z)))$
⟨*proof*⟩

**lemma** *sep-dom2*[*rule-format*]:
$x \in dom\ (C\ (list2FWpolicy\ (separate\ p))) \longrightarrow x \in dom\ (C\ (list2FWpolicy(\ p)))$
⟨*proof*⟩

**lemma** *sepDom*: $dom\ (C\ (list2FWpolicy\ p)) = dom\ (C\ (list2FWpolicy\ (separate\ p)))$
⟨*proof*⟩

**lemma** *C-eq-s-ext*[*rule-format*]: $p \neq [] \longrightarrow$
$C\ (list2FWpolicy\ (separate\ p))\ a\ = C\ (list2FWpolicy\ p)\ a$
⟨*proof*⟩

**lemma** *C-eq-s*: $p \neq [] \Longrightarrow C\ (list2FWpolicy\ (separate\ p)) = C\ (list2FWpolicy\ p)$
⟨*proof*⟩

**lemma** *sortnMTQ*: $p \neq [] \Longrightarrow qsort\ p\ l \neq []$
⟨*proof*⟩

**lemmas** *C-eq-Lemmas-sep* =
*C-eq-Lemmas sortnMT sortnMTQ RS2-NMT NMPrd NMPRS3*

**lemma** *C-eq-until-separated*:
$[\![DenyAll \in set\ (policy2list\ p);\ all\text{-}in\text{-}list\ (policy2list\ p)\ l;$
$allNetsDistinct\ (policy2list\ p)]\!] \Longrightarrow$
$C\ (list2FWpolicy\ (separate\ (sort\ (removeShadowRules2\ (remdups\ (removeShadowRules3\ C$
$(insertDeny\ (removeShadowRules1\ (policy2list\ p))))))\ l))) = C\ p$
⟨*proof*⟩

**lemma** *C-eq-until-separatedQ*:
$[\![DenyAll \in set\ (policy2list\ p);\ all\text{-}in\text{-}list\ (policy2list\ p)\ l;$
$allNetsDistinct\ (policy2list\ p)]\!] \Longrightarrow$
$C\ (list2FWpolicy\ (separate\ (qsort\ (removeShadowRules2\ (remdups\ (removeShadowRules3\ C$
$(insertDeny\ (removeShadowRules1\ (policy2list\ p))))))\ l))) = C\ p$
⟨*proof*⟩

**lemma** *domID*[*rule-format*]: $p \neq [] \wedge x \in dom(C(list2FWpolicy\ p)) \longrightarrow$
$x \in dom\ (C(list2FWpolicy(insertDenies\ p)))$
⟨*proof*⟩

**lemma** *DA-is-deny*:

$x \in dom\ (C\ (DenyAllFromTo\ a\ b \oplus DenyAllFromTo\ b\ a \oplus DenyAllFromTo\ a\ b)) \Longrightarrow$
$C\ (DenyAllFromTo\ a\ b \oplus DenyAllFromTo\ b\ a \oplus DenyAllFromTo\ a\ b)\ x = Some\ (deny\ ())$
⟨*proof*⟩

**lemma** *iDdomAux*[*rule-format*]:
$p \neq [] \longrightarrow x \notin dom\ (C\ (list2FWpolicy\ p)) \longrightarrow$
$x \in dom\ (C\ (list2FWpolicy\ (insertDenies\ p))) \longrightarrow$
$C\ (list2FWpolicy\ (insertDenies\ p))\ x = Some\ (deny\ ())$
⟨*proof*⟩

**lemma** *iD-isD*[*rule-format*]: $p \neq [] \longrightarrow x \notin dom\ (C\ (list2FWpolicy\ p))$
$\longrightarrow C\ (DenyAll \oplus list2FWpolicy\ (insertDenies\ p))\ x = C\ DenyAll\ x$
⟨*proof*⟩

**lemma** *inDomConc*:$[\![ x \notin dom\ (C\ a);\ x \notin dom\ (C\ (list2FWpolicy\ p))]\!] \Longrightarrow$
$x \notin dom\ (C\ (list2FWpolicy(a\#p)))$
⟨*proof*⟩

**lemma** *domsdisj*[*rule-format*]: $p \neq [] \longrightarrow (\forall\ x\ s.\ s \in set\ p \wedge x \in dom\ (C\ A) \longrightarrow$
$x \notin dom\ (C\ s)) \longrightarrow y \in dom\ (C\ A) \longrightarrow$
$y \notin dom\ (C\ (list2FWpolicy\ p))$
⟨*proof*⟩

**lemma** *isSepaux*:
$[\![p \neq []; noDenyAll\ (a\#p);\ separated\ (a\ \#\ p);$
$x \in dom\ (C\ (DenyAllFromTo\ (first\text{-}srcNet\ a)\ (first\text{-}destNet\ a) \oplus$
$DenyAllFromTo\ (first\text{-}destNet\ a)\ (first\text{-}srcNet\ a) \oplus a))]\!] \Longrightarrow$
$x \notin dom\ (C\ (list2FWpolicy\ p))$
⟨*proof*⟩

**lemma** *noneMTsep*[*rule-format*]: $noneMT\ C\ p \longrightarrow noneMT\ C\ (separate\ p)$
⟨*proof*⟩

**lemma** *dom-id*:
$[\![noDenyAll\ (a\#p);\ separated\ (a\#p);\ p \neq [];\ x \notin dom\ (C\ (list2FWpolicy\ p));$
$x \in dom\ (C\ (a))]\!]$
$\Longrightarrow x \notin dom\ (C\ (list2FWpolicy\ (insertDenies\ p)))$
⟨*proof*⟩

**lemma** *C-eq-iD-aux2*[*rule-format*]:
$noDenyAll1\ p \longrightarrow$
$separated\ p \longrightarrow$
$p \neq [] \longrightarrow$
$x \in dom\ (C\ (list2FWpolicy\ p)) \longrightarrow$

$C(list2FWpolicy\ (insertDenies\ p))\ x = C(list2FWpolicy\ p)\ x$
⟨*proof*⟩

**lemma** *C-eq-iD*: ⟦*separated p*; *noDenyAll1 p*; *wellformed-policy1-strong p*⟧ ⟹
*C* (*list2FWpolicy* (*insertDenies p*)) = *C* (*list2FWpolicy p*)
⟨*proof*⟩

**lemma** *noDAsortQ*[*rule-format*]: *noDenyAll1 p* ⟶ *noDenyAll1* (*qsort p l*)
⟨*proof*⟩

**lemma** *NetsCollectedSortQ*: *distinct p* ⟹*noDenyAll1 p* ⟹ *all-in-list p l* ⟹
*singleCombinators p* ⟹ *NetsCollected* (*qsort p l*)
⟨*proof*⟩

**lemmas** *CLemmas* = *nMTSort nMTSortQ noneMTRS2 noneMTrd*
*noDAsort noDAsortQ nDASC wp1-eq wp1ID*
*SCp2l ANDSep wp1n-RS2*
*OTNSEp OTNSC noDA1sep wp1-alternativesep wellformed-eq*
*wellformed1-alternative-sorted*

**lemmas** *C-eqLemmas-id* = *CLemmas NC2Sep NetsCollectedSep*
*NetsCollectedSort NetsCollectedSortQ separatedNC*

**lemma** *C-eq-Until-InsertDenies*: ⟦*DenyAll* ∈ *set* (*policy2list p*); *all-in-list*
(*policy2list p*) *l*; *allNetsDistinct* (*policy2list p*)⟧ ⟹
*C* (*list2FWpolicy* ((*insertDenies* (*separate* (*sort* (*removeShadowRules2* (*remdups*
(*removeShadowRules3 C* (*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) *l*))))) =
*C p*
⟨*proof*⟩

**lemma** *C-eq-Until-InsertDeniesQ*: ⟦*DenyAll* ∈ *set* (*policy2list p*); *all-in-list*
(*policy2list p*) *l*; *allNetsDistinct* (*policy2list p*)⟧ ⟹
*C* (*list2FWpolicy* ((*insertDenies* (*separate* (*qsort* (*removeShadowRules2* (*remdups*
(*removeShadowRules3 C* (*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) *l*))))) =
*C p*
⟨*proof*⟩

**lemma** *C-eq-RD-aux*[*rule-format*]: *C* (*p*) *x* = *C* (*removeDuplicates p*) *x*
⟨*proof*⟩

**lemma** *C-eq-RAD-aux*[*rule-format*]:
$p \neq [] \longrightarrow C\ (list2FWpolicy\ p)\ x = C\ (list2FWpolicy\ (removeAllDuplicates\ p))\ x$
⟨*proof*⟩

**lemma** *C-eq-RAD*:
$p \neq [] \Longrightarrow C\ (list2FWpolicy\ p) = C\ (list2FWpolicy\ (removeAllDuplicates\ p))$
⟨*proof*⟩

**lemma** *C-eq-compile*:
$[\![DenyAll \in set\ (policy2list\ p);\ all\text{-}in\text{-}list\ (policy2list\ p)\ l;$
$allNetsDistinct\ (policy2list\ p)]\!] \Longrightarrow$
$C\ (list2FWpolicy\ (removeAllDuplicates\ (insertDenies\ (separate\ (sort$
$(removeShadowRules2\ (remdups\ (removeShadowRules3\ C\ (insertDeny$
$(removeShadowRules1\ (policy2list\ p))))))\ l))))) = C\ p$
⟨*proof*⟩

**lemma** *C-eq-compileQ*:
$[\![DenyAll \in set\ (policy2list\ p);\ all\text{-}in\text{-}list\ (policy2list\ p)\ l;$
$allNetsDistinct\ (policy2list\ p)]\!] \Longrightarrow$
$C\ (list2FWpolicy\ (removeAllDuplicates\ (insertDenies\ (separate\ (qsort$
$(removeShadowRules2\ (remdups\ (removeShadowRules3\ C\ (insertDeny$
$(removeShadowRules1\ (policy2list\ p))))))\ l))))) = C\ p$
⟨*proof*⟩

**lemma** *C-eq-normalize*:
$[\![DenyAll \in set\ (policy2list\ p);$
$allNetsDistinct\ (policy2list\ p);$
$all\text{-}in\text{-}list\ (policy2list\ p)\ (Nets\text{-}List\ p)]\!] \Longrightarrow$
$C\ (list2FWpolicy\ (normalize\ p)) = C\ p$
⟨*proof*⟩

**lemma** *C-eq-normalizeQ*:
$[\![DenyAll \in set\ (policy2list\ p);$
$allNetsDistinct\ (policy2list\ p);$
$all\text{-}in\text{-}list\ (policy2list\ p)\ (Nets\text{-}List\ p)]\!] \Longrightarrow$
$C\ (list2FWpolicy\ (normalizeQ\ p)) = C\ p$
⟨*proof*⟩

**lemma** *domSubset3*: $dom\ (C\ (DenyAll \oplus x)) = dom\ (C\ (DenyAll))$
⟨*proof*⟩

**lemma** *domSubset4*: $dom\ (C\ (DenyAllFromTo\ x\ y \oplus DenyAllFromTo\ y\ x \oplus AllowPortFromTo\ x\ y\ dn)) =$
$dom\ (C\ (DenyAllFromTo\ x\ y \oplus DenyAllFromTo\ y\ x))$
⟨*proof*⟩

**lemma** *domSubset5*:
$dom\ (C\ (DenyAllFromTo\ x\ y \oplus DenyAllFromTo\ y\ x \oplus AllowPortFromTo\ y\ x\ dn)) =$
$dom\ (C\ (DenyAllFromTo\ x\ y \oplus DenyAllFromTo\ y\ x))$
⟨*proof*⟩

**lemma** *domSubset1*: $dom\ (C\ (DenyAllFromTo\ one\ two \oplus DenyAllFromTo\ two\ one \oplus AllowPortFromTo\ one\ two\ dn \oplus x)) =$
$dom\ (C\ (DenyAllFromTo\ one\ two \oplus DenyAllFromTo\ two\ one \oplus x))$
⟨*proof*⟩

**lemma** *domSubset2*:
$dom\ (C\ (DenyAllFromTo\ one\ two \oplus DenyAllFromTo\ two\ one \oplus AllowPortFromTo\ two\ one\ dn \oplus x)) =$
$dom\ (C\ (DenyAllFromTo\ one\ two \oplus DenyAllFromTo\ two\ one \oplus x))$
⟨*proof*⟩

**lemma** *ConcAssoc2*: $C\ (X \oplus Y \oplus ((A \oplus B) \oplus D)) = C\ (X \oplus Y \oplus A \oplus B \oplus D)$
⟨*proof*⟩

**lemma** *ConcAssoc3*: $C\ (X \oplus ((Y \oplus A) \oplus D)) = C\ (X \oplus Y \oplus A \oplus D)$
⟨*proof*⟩

**lemma** *RS3-NMT*[*rule-format*]: $DenyAll \in set\ p \longrightarrow$
$removeShadowRules3\ C\ p \neq []$
⟨*proof*⟩

**lemma** *norm-notMT*: $DenyAll \in set\ (policy2list\ p) \Longrightarrow normalize\ p \neq []$
⟨*proof*⟩

**lemma** *norm-notMTQ*: $DenyAll \in set\ (policy2list\ p) \Longrightarrow normalizeQ\ p \neq []$
⟨*proof*⟩

**lemma** *domDA*: $dom\ (C\ (DenyAll \oplus A)) = dom\ (C\ (DenyAll))$
⟨*proof*⟩

**lemmas** *domain-reasoning = domDA ConcAssoc2 domSubset1 domSubset2 domSubset3 domSubset4 domSubset5 domSubsetDistr1 domSubsetDistr2 domSubsetDistrA domSubsetDistrD coerc-assoc ConcAssoc ConcAssoc3*

The following lemmas help with the normalisation

**lemma** *list2policyR-Start*[*rule-format*]: $p \in dom\ (C\ a) \longrightarrow C\ (list2policyR\ (a\ \#\ list))\ p = C\ a\ p$
⟨*proof*⟩

**lemma** *list2policyR-End*: $p \notin dom\ (C\ a) \Longrightarrow C\ (list2policyR\ (a\ \#\ list))\ p = (C\ a \bigoplus list2policy\ (map\ C\ list))\ p$
⟨*proof*⟩

**lemma** *l2polR-eq-el*[*rule-format*]: $N \neq [] \longrightarrow C(\ list2policyR\ N)\ p = (list2policy\ (map\ C\ N))\ p$
⟨*proof*⟩

**lemma** *l2polR-eq*: $N \neq [] \Longrightarrow C(\ list2policyR\ N) = (list2policy\ (map\ C\ N))$
⟨*proof*⟩

**lemma** *list2FWpolicys-eq-el*[*rule-format*]:
$Filter \neq [] \longrightarrow C\ (list2policyR\ Filter)\ p = C\ (list2FWpolicy\ (rev\ Filter))\ p$
⟨*proof*⟩

**lemma** *list2FWpolicys-eq*:
$Filter \neq [] \Longrightarrow C\ (list2policyR\ Filter) = C\ (list2FWpolicy\ (rev\ Filter))$
⟨*proof*⟩

**lemma** *list2FWpolicys-eq-sym*:
$Filter \neq [] \Longrightarrow C\ (list2policyR\ (rev\ Filter)) = C\ (list2FWpolicy\ Filter)$
⟨*proof*⟩

**lemma** *p-eq*[*rule-format*]: $p \neq [] \longrightarrow$
$list2policy\ (map\ C\ (rev\ p)) = C\ (list2FWpolicy\ p)$
⟨*proof*⟩

**lemma** *p-eq2*[*rule-format*]: $normalize\ x \neq [] \longrightarrow$
$C\ (list2FWpolicy\ (normalize\ x)) = C\ x \longrightarrow$
$list2policy\ (map\ C\ (rev\ (normalize\ x))) = C\ x$
⟨*proof*⟩

**lemma** *p-eq2Q*[*rule-format*]: $normalizeQ\ x \neq [] \longrightarrow$
$C\ (list2FWpolicy\ (normalizeQ\ x)) = C\ x \longrightarrow$
$list2policy\ (map\ C\ (rev\ (normalizeQ\ x))) = C\ x$
⟨*proof*⟩

**lemma** *list2listNMT*[*rule-format*]: $x \neq [] \longrightarrow map\ sem\ x \neq []$
⟨*proof*⟩

**lemma** *Norm-Distr2*:
$r\ o\text{-}f\ ((P \bigotimes_2 (list2policy\ Q))\ o\ d) =$
$(list2policy\ ((P \bigotimes_L Q)\ (op \bigotimes_2)\ r\ d))$
⟨*proof*⟩

**lemma** *NATDistr*: $[\![N \neq []; F = C\ (list2policyR\ N)]\!] \Longrightarrow$
$((\lambda\ (x,y).\ x)\ o\text{-}f\ ((NAT \bigotimes_2 F)\ o\ (\lambda\ x.\ (x,x))) =$
$(list2policy\ (\ ((NAT \bigotimes_L (map\ C\ N))\ (op \bigotimes_2)$
$(\lambda\ (x,y).\ x)\ (\lambda\ x.\ (x,x))))))$
⟨*proof*⟩

**lemma** *C-eq-normalize-manual*:
$[\![DenyAll \in set\ (policy2list\ p);$
$allNetsDistinct\ (policy2list\ p);$
$all\text{-}in\text{-}list\ (policy2list\ p)\ l]\!] \Longrightarrow$
$C\ (list2FWpolicy\ (normalize\text{-}manual\text{-}order\ p\ l)) = C\ p$
⟨*proof*⟩

**lemma** *p-eq2-manualQ*[*rule-format*]: $normalize\text{-}manual\text{-}orderQ\ x\ l \neq [] \longrightarrow$
$C\ (list2FWpolicy\ (normalize\text{-}manual\text{-}orderQ\ x\ l)) = C\ x \longrightarrow$

*list2policy* (*map C* (*rev* (*normalize-manual-orderQ x l*))) = *C x*
⟨*proof*⟩

**lemma** *norm-notMT-manualQ*: *DenyAll* ∈ *set* (*policy2list p*) ⟹ *normalize-manual-orderQ p l* ≠ []
⟨*proof*⟩

**lemma** *C-eq-normalize-manualQ*:
⟦*DenyAll* ∈ *set* (*policy2list p*);
*allNetsDistinct* (*policy2list p*);
*all-in-list* (*policy2list p*) *l*⟧ ⟹
*C* (*list2FWpolicy* (*normalize-manual-orderQ p l*)) = *C p*
⟨*proof*⟩

**lemma** *p-eq2-manual*[*rule-format*]: *normalize-manual-order x l* ≠ [] ⟶
*C* (*list2FWpolicy* (*normalize-manual-order x l*)) = *C x* ⟶
*list2policy* (*map C* (*rev* (*normalize-manual-order x l*))) = *C x*
⟨*proof*⟩

**lemma** *norm-notMT-manual*: *DenyAll* ∈ *set* (*policy2list p*) ⟹ *normalize-manual-order p l* ≠ []
⟨*proof*⟩

As an example, how this theorems can be used for a concrete normalisation instantiation.

**lemma** *normalizeNAT*: ⟦*DenyAll* ∈ *set* (*policy2list Filter*);
*allNetsDistinct* (*policy2list Filter*);
*all-in-list* (*policy2list Filter*) (*Nets-List Filter*)⟧ ⟹
((λ (*x*,*y*). *x*) *o-f* (((*NAT* ⨂$_2$ *C Filter*) *o* (λ*x*. (*x*,*x*))))) =
*list2policy* ((*NAT* ⨂$_L$ (*map C* (*rev* (*normalize Filter*)))) (*op* ⨂$_2$) (λ (*x*,*y*). *x*) (λ *x*. (*x*,*x*)))
⟨*proof*⟩

**lemma** *domSimpl*[*simp*]: *dom* (*C* (*A* ⊕ *DenyAll*)) = *dom* (*C* (*DenyAll*))
⟨*proof*⟩

The followin theorems can be applied when prepending the usual normalisation with an additional step and using another semantical interpretation function. This is a general recipe which can be applied whenever one nees to combine several normalisation strategies.

**lemma** *CRotate-eq-rotateC*: *CRotate p* = *C* (*rotatePolicy p*)

⟨*proof*⟩

**lemma** *DAinRotate*: ⟦*DenyAll* ∈ *set* (*policy2list p*)⟧
⟹ *DenyAll* ∈ *set* (*policy2list* (*rotatePolicy p*))
⟨*proof*⟩

**lemma** *DAUniv*: *dom* (*CRotate* (*P* ⊕ *DenyAll*)) = *UNIV*
⟨*proof*⟩

**lemma** *p-eq2R*[*rule-format*]: *normalize* (*rotatePolicy x*) ≠ [] ⟶
*C* (*list2FWpolicy* (*normalize* (*rotatePolicy x*))) = *CRotate x* ⟶
*list2policy* (*map C* (*rev* (*normalize* (*rotatePolicy x*)))) = *CRotate x*
⟨*proof*⟩

**lemma** *C-eq-normalizeRotate*:
⟦*DenyAll* ∈ *set* (*policy2list p*);
*allNetsDistinct* (*policy2list* (*rotatePolicy p*));
*all-in-list* (*policy2list* (*rotatePolicy p*)) (*Nets-List* (*rotatePolicy p*))⟧ ⟹
*C* (*list2FWpolicy* (*removeAllDuplicates* (*insertDenies* (*separate* (*sort*
(*removeShadowRules2* (*remdups* (*removeShadowRules3 C* (*insertDeny*
(*removeShadowRules1* (*policy2list* (*rotatePolicy p*)))))))
(*Nets-List* (*rotatePolicy p*))))))) = *CRotate p*
⟨*proof*⟩

**lemma** *C-eq-normalizeRotate2*:
⟦*DenyAll* ∈ *set* (*policy2list p*);
*allNetsDistinct* (*policy2list* (*rotatePolicy p*));
*all-in-list* (*policy2list* (*rotatePolicy p*)) (*Nets-List* (*rotatePolicy p*))⟧ ⟹
*C* (*list2FWpolicy* (*normalize* (*rotatePolicy p*))) = *CRotate p*
⟨*proof*⟩

**end**

**theory** *NormalisationIPPProofs*
**imports** *NormalisationIntegerPortProof*
**begin**

Normalisation proofs which are specific to the IntegerProtocol address representation.

**lemma** *ConcAssoc*: $Cp((A \oplus B) \oplus D) = Cp(A \oplus (B \oplus D))$
⟨*proof*⟩

**lemma** *aux26*[*simp*]: *twoNetsDistinct a b c d* ⟹
$dom\ (Cp\ (AllowPortFromTo\ a\ b\ p)) \cap dom\ (Cp\ (DenyAllFromTo\ c\ d)) = \{\}$
⟨*proof*⟩

**lemma** *wp2-aux*[*rule-format*]: *wellformed-policy2Pr* $(xs\ @\ [x]) \longrightarrow$
*wellformed-policy2Pr xs*
⟨*proof*⟩

**lemma** *Cdom2*: $x \in dom(Cp\ b) \Longrightarrow Cp\ (a \oplus b)\ x = (Cp\ b)\ x$
⟨*proof*⟩

**lemma** *wp2Conc*[*rule-format*]: *wellformed-policy2Pr* $(x\#xs) \Longrightarrow$ *wellformed-policy2Pr xs*
⟨*proof*⟩

**lemma** *DAimpliesMR-E*[*rule-format*]: *DenyAll* $\in$ *set p* $\longrightarrow$
$(\exists\ r.$ *applied-rule-rev Cp x p* $=$ *Some* $r)$
⟨*proof*⟩

**lemma** *DAimplieMR*[*rule-format*]: *DenyAll* $\in$ *set p* ⟹ *applied-rule-rev Cp x p* $\neq$ *None*
⟨*proof*⟩

**lemma** *MRList1*[*rule-format*]: $x \in dom\ (Cp\ a) \Longrightarrow$ *applied-rule-rev Cp x* $(b@[a]) =$ *Some a*
⟨*proof*⟩

**lemma** *MRList2*: $x \in dom\ (Cp\ a) \Longrightarrow$ *applied-rule-rev Cp x* $(c@b@[a]) =$ *Some a*
⟨*proof*⟩

**lemma** *MRList3*: $x \notin dom\ (Cp\ xa) \Longrightarrow$
*applied-rule-rev Cp x* $(a\ @\ b\ \#\ xs\ @\ [xa]) =$ *applied-rule-rev Cp x* $(a\ @\ b\ \#\ xs)$
⟨*proof*⟩

**lemma** *CConcEnd*[*rule-format*]: *Cp a x* $=$ *Some y* $\longrightarrow$
*Cp* $(list2FWpolicy\ (xs\ @\ [a]))\ x =$ *Some y*
(**is** *?P xs*)
⟨*proof*⟩

**lemma** *CConcStartaux*: $[\![Cp\ a\ x = None]\!] \Longrightarrow (Cp\ aa\ ++\ Cp\ a)\ x = Cp\ aa\ x$
⟨*proof*⟩

**lemma** *CConcStart*[*rule-format*]: $xs \neq [] \longrightarrow Cp\ a\ x = None \longrightarrow$
*Cp* $(list2FWpolicy\ (xs\ @\ [a]))\ x = Cp\ (list2FWpolicy\ xs)\ x$
⟨*proof*⟩

**lemma** *mrNnt*[*simp*]: *applied-rule-rev Cp x p = Some a* $\Longrightarrow$ *p* $\neq$ []
⟨*proof*⟩

**lemma** *mr-is-C*[*rule-format*]: *applied-rule-rev Cp x p = Some a* $\longrightarrow$
*Cp* (*list2FWpolicy* (*p*)) *x* = *Cp a x*
⟨*proof*⟩

**lemma** *CConcStart2*: ⟦*p* $\neq$ []; *x* $\notin$ *dom* (*Cp a*)⟧ $\Longrightarrow$
*Cp* (*list2FWpolicy* (*p*@[*a*])) *x* = *Cp* (*list2FWpolicy p*)*x*
⟨*proof*⟩

**lemma** *CConcEnd1*: ⟦*q*@*p* $\neq$ []; *x* $\notin$ *dom* (*Cp a*)⟧ $\Longrightarrow$
*Cp* (*list2FWpolicy* (*q*@*p*@[*a*])) *x* = *Cp* (*list2FWpolicy* (*q*@*p*))*x*
⟨*proof*⟩

**lemma** *CConcEnd2*[*rule-format*]: *x* $\in$ *dom* (*Cp a*) $\longrightarrow$
*Cp* (*list2FWpolicy* (*xs* @ [*a*])) *x* = *Cp a x*
(**is** *?P xs*)
⟨*proof*⟩

**lemma** *bar3*: *x* $\in$ *dom* (*Cp* (*list2FWpolicy* (*xs* @ [*xa*]))) $\Longrightarrow$
*x* $\in$ *dom* (*Cp* (*list2FWpolicy xs*)) $\vee$ *x* $\in$ *dom* (*Cp xa*)
⟨*proof*⟩

**lemma** *CeqEnd*[*rule-format*,*simp*]: *a* $\neq$ [] $\longrightarrow$ *x* $\in$ *dom* (*Cp* (*list2FWpolicy a*)) $\longrightarrow$
*Cp* (*list2FWpolicy* (*b*@*a*)) *x* = (*Cp* (*list2FWpolicy a*)) *x*
⟨*proof*⟩

**lemma** *CConcStartA*[*rule-format*,*simp*]: *x* $\in$ *dom* (*Cp a*) $\longrightarrow$
*x* $\in$ *dom* (*Cp* (*list2FWpolicy* (*a* # *b*)))
(**is** *?P b*)
⟨*proof*⟩

**lemma** *domConc*: ⟦*x* $\in$ *dom* (*Cp* (*list2FWpolicy b*)); *b* $\neq$ []⟧ $\Longrightarrow$
*x* $\in$ *dom* (*Cp* (*list2FWpolicy* (*a*@*b*)))
⟨*proof*⟩

**lemma** *CeqStart*[*rule-format*,*simp*]:
*x* $\notin$ *dom* (*Cp* (*list2FWpolicy a*)) $\longrightarrow$ *a* $\neq$ [] $\longrightarrow$ *b* $\neq$ [] $\longrightarrow$
*Cp* (*list2FWpolicy* (*b*@*a*)) *x* = (*Cp* (*list2FWpolicy b*)) *x*
⟨*proof*⟩

**lemma** *C-eq-if-mr-eq2*: ⟦*applied-rule-rev Cp x a = Some r*; *applied-rule-rev Cp x b = Some r*;
*a* $\neq$ []; *b* $\neq$ []⟧ $\Longrightarrow$

$(Cp\ (list2FWpolicy\ a))\ x = (Cp\ (list2FWpolicy\ b))\ x$

⟨*proof*⟩

**lemma** *nMRtoNone*[*rule-format*]: $p \neq [] \longrightarrow applied\text{-}rule\text{-}rev\ Cp\ x\ p = None \longrightarrow$
$Cp\ (list2FWpolicy\ p)\ x = None$

⟨*proof*⟩

**lemma** *C-eq-if-mr-eq*:
$[\![applied\text{-}rule\text{-}rev\ Cp\ x\ b = applied\text{-}rule\text{-}rev\ Cp\ x\ a;\ a \neq [];\ b \neq []]\!] \Longrightarrow$
$(Cp\ (list2FWpolicy\ a))\ x = (Cp\ (list2FWpolicy\ b))\ x$

⟨*proof*⟩

**lemma** *notmatching-notdom*: $applied\text{-}rule\text{-}rev\ Cp\ x\ (p@[a]) \neq Some\ a \Longrightarrow x \notin dom\ (Cp\ a)$

⟨*proof*⟩

**lemma** *foo3a*[*rule-format*]: $applied\text{-}rule\text{-}rev\ Cp\ x\ (a@[b]@c) = Some\ b \longrightarrow\ r \in set\ c \longrightarrow$
$b \notin set\ c \longrightarrow x \notin dom\ (Cp\ r)$

⟨*proof*⟩

**lemma** *foo3D*: $[\![wellformed\text{-}policy1\ p;\ p = (DenyAll\#ps);$
$applied\text{-}rule\text{-}rev\ Cp\ x\ p = Some\ DenyAll;\ r \in set\ ps]\!] \Longrightarrow x \notin dom\ (Cp\ r)$

⟨*proof*⟩

**lemma** *foo4*[*rule-format*]: $set\ p = set\ s \wedge (\forall\ r.\ r \in set\ p \longrightarrow x \notin dom\ (Cp\ r)) \longrightarrow$
$(\forall\ r\ .r \in set\ s \longrightarrow x \notin dom\ (Cp\ r))$

⟨*proof*⟩

**lemma** *foo5b*[*rule-format*]: $x \in dom\ (Cp\ b) \longrightarrow (\forall\ r.\ r \in set\ c \longrightarrow x \notin dom\ (Cp\ r)) \longrightarrow$
$applied\text{-}rule\text{-}rev\ Cp\ x\ (b\#c) = Some\ b$

⟨*proof*⟩

**lemma** *mr-first*: $[\![x \in dom\ (Cp\ b);\ (\forall\ r.\ r \in set\ c \longrightarrow x \notin dom\ (Cp\ r));\ s = b\#c]\!] \Longrightarrow$
$applied\text{-}rule\text{-}rev\ Cp\ x\ s = Some\ b$

⟨*proof*⟩

**lemma** *mr-charn*[*rule-format*]: $a \in set\ p \longrightarrow (x \in dom\ (Cp\ a)) \longrightarrow$
$(\forall\ r.\ r \in set\ p \wedge x \in dom\ (Cp\ r) \longrightarrow r = a) \longrightarrow$
$applied\text{-}rule\text{-}rev\ Cp\ x\ p = Some\ a$

⟨*proof*⟩

**lemma** *foo8*: $[\![(\forall\ r.\ r \in set\ p \wedge x \in dom\ (Cp\ r) \longrightarrow r = a);\ set\ p = set\ s]\!] \Longrightarrow$
$(\forall\ r.\ r \in set\ s \wedge x \in dom\ (Cp\ r) \longrightarrow r = a)$

⟨*proof*⟩

**lemma** *mrConcEnd*[*rule-format*]: $applied\text{-}rule\text{-}rev\ Cp\ x\ (b\ \#\ p) = Some\ a \longrightarrow a \neq b \longrightarrow$
$applied\text{-}rule\text{-}rev\ Cp\ x\ p = Some\ a$

⟨*proof*⟩

**lemma** *wp3tl*[*rule-format*]: *wellformed-policy3Pr p* $\longrightarrow$ *wellformed-policy3Pr* (*tl p*)
⟨*proof*⟩

**lemma** *wp3Conc*[*rule-format*]: *wellformed-policy3Pr* (*a*#*p*) $\longrightarrow$ *wellformed-policy3Pr p*
⟨*proof*⟩

**lemma** *foo98*[*rule-format*]:*applied-rule-rev Cp x* (*aa* # *p*) = *Some a* $\longrightarrow$ *x* $\in$ *dom* (*Cp r*) $\longrightarrow$
*r* $\in$ *set p*
$\longrightarrow$ *a* $\in$ *set p*
⟨*proof*⟩

**lemma** *mrMTNone*[*simp*]: *applied-rule-rev Cp x* [] = *None*
⟨*proof*⟩

**lemma** *DAAux*[*simp*]: *x* $\in$ *dom* (*Cp DenyAll*)
⟨*proof*⟩

**lemma** *mrSet*[*rule-format*]: *applied-rule-rev Cp x p* = *Some r* $\longrightarrow$ *r* $\in$ *set p*
⟨*proof*⟩

**lemma** *mr-not-Conc*: *singleCombinators p* $\Longrightarrow$ *applied-rule-rev Cp x p* $\neq$ *Some* (*a*$\oplus$*b*)
⟨*proof*⟩

**lemma** *foo25*[*rule-format*]: *wellformed-policy3Pr* (*p*@[*x*]) $\longrightarrow$ *wellformed-policy3Pr p*
⟨*proof*⟩

**lemma** *mr-in-dom*[*rule-format*]: *applied-rule-rev Cp x p* = *Some a* $\longrightarrow$ *x* $\in$ *dom* (*Cp a*)
⟨*proof*⟩

**lemma** *wp3EndMT*[*rule-format*]: *wellformed-policy3Pr* (*p*@[*xs*]) $\longrightarrow$
*AllowPortFromTo a b po* $\in$ *set p* $\longrightarrow$
*dom* (*Cp* (*AllowPortFromTo a b po*)) $\cap$ *dom* (*Cp xs*) = {}
⟨*proof*⟩

**lemma** *foo29*: ⟦*dom* (*Cp a*) $\neq$ {}; *dom* (*Cp a*) $\cap$ *dom* (*Cp b*) = {}⟧ $\Longrightarrow$ *a* $\neq$ *b*
⟨*proof*⟩

**lemma** *foo28*: ⟦*AllowPortFromTo a b po* $\in$ *set p*;
*dom* (*Cp* (*AllowPortFromTo a b po*)) $\neq$ {}; (*wellformed-policy3Pr* (*p*@[*x*]))⟧
$\Longrightarrow$ *x* $\neq$ *AllowPortFromTo a b po*
⟨*proof*⟩

**lemma** *foo28a*[*rule-format*]: *x* $\in$ *dom* (*Cp a*) $\Longrightarrow$ *dom* (*Cp a*) $\neq$ {}

⟨*proof*⟩

**lemma** *allow-deny-dom*[*simp*]: *dom* (*Cp* (*AllowPortFromTo a b po*)) ⊆
*dom* (*Cp* (*DenyAllFromTo a b*))
⟨*proof*⟩

**lemma** *DenyAllowDisj*: *dom* (*Cp* (*AllowPortFromTo a b p*)) ≠ {} ⟹
*dom* (*Cp* (*DenyAllFromTo a b*)) ∩ *dom* (*Cp* (*AllowPortFromTo a b p*)) ≠ {}
⟨*proof*⟩

**lemma** *foo31*: ⟦(∀ *r*. *r* ∈ *set p* ∧ *x* ∈ *dom* (*Cp r*) ⟶
(*r* = *AllowPortFromTo a b po* ∨ *r* = *DenyAllFromTo a b* ∨ *r* = *DenyAll*));
*set p* = *set s*⟧ ⟹
(∀ *r*. *r* ∈ *set s* ∧ *x* ∈ *dom* (*Cp r*) ⟶
(*r* = *AllowPortFromTo a b po* ∨ *r* = *DenyAllFromTo a b* ∨ *r* = *DenyAll*))
⟨*proof*⟩

**lemma** *wp1-auxa*: *wellformed-policy1-strong p*⟹(∃ *r*. *applied-rule-rev Cp x p* = *Some r*)
⟨*proof*⟩

**lemma** *deny-dom*[*simp*]: *twoNetsDistinct a b c d* ⟹ *dom* (*Cp* (*DenyAllFromTo a b*)) ∩
*dom* (*Cp* (*DenyAllFromTo c d*)) = {}
⟨*proof*⟩

**lemma** *domTrans*: ⟦*dom a* ⊆ *dom b*; *dom*(*b*) ∩ *dom* (*c*) = {}⟧ ⟹ *dom*(*a*) ∩ *dom*(*c*) = {}
⟨*proof*⟩

**lemma** *DomInterAllowsMT*: ⟦ *twoNetsDistinct a b c d*⟧ ⟹
*dom* (*Cp* (*AllowPortFromTo a b p*)) ∩ *dom* (*Cp* (*AllowPortFromTo c d po*)) = {}
⟨*proof*⟩

**lemma** *DomInterAllowsMT-Ports*: ⟦*p* ≠ *po*⟧ ⟹
*dom* (*Cp* (*AllowPortFromTo a b p*)) ∩ *dom* (*Cp* (*AllowPortFromTo c d po*)) = {}
⟨*proof*⟩

**lemma** *wellformed-policy3-charn*[*rule-format*]:
*singleCombinators p* ⟶ *distinct p* ⟶ *allNetsDistinct p* ⟶
*wellformed-policy1 p* ⟶ *wellformed-policy2Pr p* ⟶ *wellformed-policy3Pr p*
⟨*proof*⟩

**lemma** *DistinctNetsDenyAllow*:

⟦*DenyAllFromTo b c* ∈ *set p*; *AllowPortFromTo a d po* ∈ *set p*; *allNetsDistinct p*;
*dom* (*Cp* (*DenyAllFromTo b c*)) ∩ *dom* (*Cp* (*AllowPortFromTo a d po*)) ≠ {}⟧
⟹ *b* = *a* ∧ *c* = *d*
⟨*proof*⟩

**lemma** *DistinctNetsAllowAllow*:
⟦*AllowPortFromTo b c poo* ∈ *set p*; *AllowPortFromTo a d po* ∈ *set p*;
*allNetsDistinct p*; *dom* (*Cp* (*AllowPortFromTo b c poo*)) ∩
*dom* (*Cp* (*AllowPortFromTo a d po*)) ≠ {}⟧
⟹ *b* = *a* ∧ *c* = *d* ∧ *poo* = *po*
⟨*proof*⟩

**lemma** *WP2RS2*[*simp*]:
⟦*singleCombinators p*;
*distinct p*;
*allNetsDistinct p*⟧ ⟹
*wellformed-policy2Pr* (*removeShadowRules2 p*)
⟨*proof*⟩

**lemma** *AD-aux*: ⟦*AllowPortFromTo a b po* ∈ *set p* ;*DenyAllFromTo c d* ∈ *set p*;
*allNetsDistinct p* ; *singleCombinators p*;
*a* ≠ *c* ∨ *b* ≠ *d*⟧
⟹ *dom* (*Cp* (*AllowPortFromTo a b po*)) ∩ *dom* (*Cp* (*DenyAllFromTo c d*)) = {}
⟨*proof*⟩

**lemma** *sorted-WP2*[*rule-format*]: *sorted p l* ⟶ *all-in-list p l* ⟶ *distinct p* ⟶
*allNetsDistinct p* ⟶ *singleCombinators p* ⟶ *wellformed-policy2Pr p*
⟨*proof*⟩

**lemma** *wellformed2-sorted*[*simp*]: ⟦*all-in-list p l*; *distinct p*; *allNetsDistinct p*;
*singleCombinators p*⟧ ⟹ *wellformed-policy2Pr* (*sort p l*)
⟨*proof*⟩

**lemma** *wellformed2-sortedQ*[*simp*]: ⟦*all-in-list p l*; *distinct p*; *allNetsDistinct p*;
*singleCombinators p*⟧ ⟹ *wellformed-policy2Pr* (*qsort p l*)
⟨*proof*⟩

**lemma** *C-DenyAll*[*simp*]: *Cp* (*list2FWpolicy* (*xs* @ [*DenyAll*])) *x* = *Some* (*deny* ())

⟨*proof*⟩

**lemma** *C-eq-RS1n*:
*Cp*(*list2FWpolicy* (*removeShadowRules1-alternative p*)) = *Cp*(*list2FWpolicy p*)
⟨*proof*⟩

**lemma** *C-eq-RS1*[*simp*]: *p* ≠ [] ⟹
*Cp*(*list2FWpolicy* (*removeShadowRules1 p*)) = *Cp*(*list2FWpolicy p*)
⟨*proof*⟩

**lemma** *EX-MR-aux*[*rule-format*]: *applied-rule-rev Cp x* (*DenyAll* # *p*) ≠ *Some DenyAll* ⟶
(∃ *y*. *applied-rule-rev Cp x p* = *Some y*)
⟨*proof*⟩

**lemma** *EX-MR* : ⟦*applied-rule-rev Cp x p* ≠ (*Some DenyAll*); *p* = *DenyAll*#*ps*⟧ ⟹
(*applied-rule-rev Cp x p* = *applied-rule-rev Cp x ps*)
⟨*proof*⟩

**lemma** *mr-not-DA*:
⟦*wellformed-policy1-strong s*; *applied-rule-rev Cp x p* = *Some* (*DenyAllFromTo a ab*);
*set p* = *set s*⟧ ⟹ *applied-rule-rev Cp x s* ≠ *Some DenyAll*
⟨*proof*⟩

**lemma** *domsMT-notND-DD*:
⟦*dom* (*Cp* (*DenyAllFromTo a b*)) ∩ *dom* (*Cp* (*DenyAllFromTo c d*)) ≠ {}⟧ ⟹
¬ *netsDistinct a c*
⟨*proof*⟩

**lemma** *domsMT-notND-DD2*:
⟦*dom* (*Cp* (*DenyAllFromTo a b*)) ∩ *dom* (*Cp* (*DenyAllFromTo c d*)) ≠ {}⟧ ⟹
¬ *netsDistinct b d*
⟨*proof*⟩

**lemma** *domsMT-notND-DD3*:
⟦*x* ∈ *dom* (*Cp* (*DenyAllFromTo a b*)); *x* ∈ *dom* (*Cp* (*DenyAllFromTo c d*))⟧ ⟹
¬ *netsDistinct a c*
⟨*proof*⟩

**lemma** *domsMT-notND-DD4*:
⟦*x* ∈ *dom* (*Cp* (*DenyAllFromTo a b*)); *x* ∈ *dom* (*Cp* (*DenyAllFromTo c d*))⟧ ⟹
¬ *netsDistinct b d*
⟨*proof*⟩

**lemma** *NetsEq-if-sameP-DD*:
⟦*allNetsDistinct p*; *u*∈ *set p*; *v*∈ *set p*; *u* = (*DenyAllFromTo a b*);
*v* = (*DenyAllFromTo c d*); *x* ∈ *dom* (*Cp* (*u*)); *x* ∈ *dom* (*Cp* (*v*))⟧ ⟹
*a* = *c* ∧ *b* = *d*
⟨*proof*⟩

**lemma** *rule-charn1*:
**assumes** *aND*: *allNetsDistinct p*
**and** *mr-is-allow*: *applied-rule-rev Cp x p* = *Some* (*AllowPortFromTo a b po*)
**and** *SC*: *singleCombinators p*
**and** *inp*: *r* ∈ *set p*
**and** *inDom*: *x* ∈ *dom* (*Cp r*)
**shows** (*r* = *AllowPortFromTo a b po* ∨ *r* = *DenyAllFromTo a b* ∨ *r* = *DenyAll*)
⟨*proof*⟩

**lemma** *noneMTsubset*[*rule-format*]: *noneMT Cp a* ⟶ *set b* ⊆ *set a* ⟶ *noneMT Cp b*
⟨*proof*⟩

**lemma** *nMTSort*: *noneMT Cp p* ⟹ *noneMT Cp* (*sort p l*)
⟨*proof*⟩

**lemma** *nMTSortQ*: *noneMT Cp p* ⟹ *noneMT Cp* (*qsort p l*)
⟨*proof*⟩

**lemma** *wp3char*[*rule-format*]: *noneMT Cp xs* ∧ *Cp* (*AllowPortFromTo a b po*) = *empty* ∧
*wellformed-policy3Pr* (*xs* @ [*DenyAllFromTo a b*]) ⟶
*AllowPortFromTo a b po* ∉ *set xs*
⟨*proof*⟩

**lemma** *wp3charn*[*rule-format*]:
**assumes** *domAllow*: *dom* (*Cp* (*AllowPortFromTo a b po*)) ≠ {}
**and** *wp3*: *wellformed-policy3Pr* (*xs* @ [*DenyAllFromTo a b*])
**shows** *allowNotInList*: *AllowPortFromTo a b po* ∉ *set xs*
⟨*proof*⟩

**lemma** *rule-charn2*:
**assumes** *aND*: *allNetsDistinct p*
**and** *wp1*: *wellformed-policy1 p*
**and** *SC*: *singleCombinators p*

**and** *wp3*: *wellformed-policy3Pr p*
**and** *allow-in-list*: *AllowPortFromTo c d po* ∈ *set p*
**and** *x-in-dom-allow*: *x* ∈ *dom* (*Cp* (*AllowPortFromTo c d po*))
**shows** *applied-rule-rev Cp x p* = *Some* (*AllowPortFromTo c d po*)
⟨*proof*⟩

**lemma** *rule-charn3*:
⟦*wellformed-policy1 p*; *allNetsDistinct p*; *singleCombinators p*;
*wellformed-policy3Pr p*; *applied-rule-rev Cp x p* = *Some* (*DenyAllFromTo c d*);
*AllowPortFromTo a b po* ∈ *set p*⟧ ⟹ *x* ∉ *dom* (*Cp* (*AllowPortFromTo a b po*))
⟨*proof*⟩

**lemma** *rule-charn4*:
**assumes** *wp1*: *wellformed-policy1 p*
**and** *aND*: *allNetsDistinct p*
**and** *SC*: *singleCombinators p*
**and** *wp3*: *wellformed-policy3Pr p*
**and** *DA*: *DenyAll* ∉ *set p*
**and** *mr*: *applied-rule-rev Cp x p* = *Some* (*DenyAllFromTo a b*)
**and** *rinp*: *r* ∈ *set p*
**and** *xindom*: *x* ∈ *dom* (*Cp r*)
**shows** *r* = *DenyAllFromTo a b*
⟨*proof*⟩

**lemma** *foo31a*: ⟦(∀ *r*. *r* ∈ *set p* ∧ *x* ∈ *dom* (*Cp r*) ⟶
(*r* = *AllowPortFromTo a b po* ∨ *r* = *DenyAllFromTo a b* ∨ *r* = *DenyAll*));
*set p* = *set s*; *r* ∈ *set s* ; *x* ∈ *dom* (*Cp r*)⟧ ⟹
(*r* = *AllowPortFromTo a b po* ∨ *r* = *DenyAllFromTo a b* ∨ *r* = *DenyAll*)
⟨*proof*⟩

**lemma** *aux4*[*rule-format*]:
*applied-rule-rev Cp x* (*a*#*p*) = *Some a* ⟶ *a* ∉ *set* (*p*) ⟶ *applied-rule-rev Cp x p* = *None*
⟨*proof*⟩

**lemma** *mrDA-tl*:
**assumes** *mr-DA*: *applied-rule-rev Cp x p* = *Some DenyAll*
**and** *wp1n*: *wellformed-policy1-strong p*
**shows** *applied-rule-rev Cp x* (*tl p*) = *None*
⟨*proof*⟩

**lemma** *rule-charnDAFT*:
⟦*wellformed-policy1-strong p*; *allNetsDistinct p*; *singleCombinators p*;
*wellformed-policy3Pr p*; *applied-rule-rev Cp x p* = *Some* (*DenyAllFromTo a b*);
*r* ∈ *set* (*tl p*); *x* ∈ *dom* (*Cp r*)⟧ ⟹ *r* = *DenyAllFromTo a b*
⟨*proof*⟩

**lemma** *mrDenyAll-is-unique*:
⟦*wellformed-policy1-strong p*; *applied-rule-rev Cp x p* = *Some DenyAll*;
*r* ∈ *set* (*tl p*)⟧ ⟹ *x* ∉ *dom* (*Cp r*)
⟨*proof*⟩

**theorem** *C-eq-Sets-mr*:
**assumes** *sets-eq*: *set p* = *set s*
**and** *SC*: *singleCombinators p*
**and** *wp1-p*: *wellformed-policy1-strong p*
**and** *wp1-s*: *wellformed-policy1-strong s*
**and** *wp3-p*: *wellformed-policy3Pr p*
**and** *wp3-s*: *wellformed-policy3Pr s*
**and** *aND*: *allNetsDistinct p*

**shows** *applied-rule-rev Cp x p* = *applied-rule-rev Cp x s*
⟨*proof*⟩

**lemma** *C-eq-Sets*:
⟦*singleCombinators p*; *wellformed-policy1-strong p*; *wellformed-policy1-strong s*;
*wellformed-policy3Pr p*; *wellformed-policy3Pr s*; *allNetsDistinct p*; *set p* = *set s*⟧ ⟹
*Cp* (*list2FWpolicy p*) *x* = *Cp* (*list2FWpolicy s*) *x*
⟨*proof*⟩

**lemma** *C-eq-sorted*: ⟦*distinct p*; *all-in-list p l*; *singleCombinators p*;
*wellformed-policy1-strong p*; *wellformed-policy3Pr p*; *allNetsDistinct p*⟧ ⟹
*Cp* (*list2FWpolicy* (*sort p l*))= *Cp* (*list2FWpolicy p*)
⟨*proof*⟩

**lemma** *C-eq-sortedQ*: ⟦*distinct p*; *all-in-list p l*; *singleCombinators p*;
*wellformed-policy1-strong p*; *wellformed-policy3Pr p*; *allNetsDistinct p*⟧ ⟹
*Cp* (*list2FWpolicy* (*qsort p l*))= *Cp* (*list2FWpolicy p*)
⟨*proof*⟩

**lemma** *C-eq-RS2-mr*: *applied-rule-rev Cp x* (*removeShadowRules2 p*)= *applied-rule-rev Cp x p*
⟨*proof*⟩

**lemma** *C-eq-None*[*rule-format*]: *p* ≠ [] −−> *applied-rule-rev Cp x p* = *None* ⟶
*Cp* (*list2FWpolicy p*) *x* = *None*
⟨*proof*⟩

**lemma** *C-eq-None2*:
$[\![a \neq []; b \neq []; applied\text{-}rule\text{-}rev\ Cp\ x\ a = None; applied\text{-}rule\text{-}rev\ Cp\ x\ b = None]\!] \Longrightarrow$
$(Cp\ (list2FWpolicy\ a))\ x = (Cp\ (list2FWpolicy\ b))\ x$
⟨*proof*⟩

**lemma** *C-eq-RS2*: *wellformed-policy1-strong p* $\Longrightarrow$
*Cp* (*list2FWpolicy* (*removeShadowRules2 p*))= *Cp* (*list2FWpolicy p*)
⟨*proof*⟩

**lemma** *noneMTRS2*: *noneMT Cp p* $\Longrightarrow$ *noneMT Cp* (*removeShadowRules2 p*)
⟨*proof*⟩

**lemma** *CconcNone*: $[\![dom\ (Cp\ a) = \{\}; p \neq []]\!] \Longrightarrow$
*Cp* (*list2FWpolicy* (*a* # *p*)) *x* = *Cp* (*list2FWpolicy p*) *x*
⟨*proof*⟩

**lemma** *noneMTrd*[*rule-format*]: *noneMT Cp p* $\longrightarrow$ *noneMT Cp* (*remdups p*)
⟨*proof*⟩

**lemma** *DARS3*[*rule-format*]:*DenyAll* $\notin$ *set p*$\longrightarrow$*DenyAll* $\notin$ *set* (*removeShadowRules3 Cp p*)
⟨*proof*⟩

**lemma** *DAnMT*: *dom* (*Cp DenyAll*) $\neq \{\}$
⟨*proof*⟩

**lemma** *DAnMT2*: *Cp DenyAll* $\neq$ *empty*
⟨*proof*⟩

**lemma** *wp1n-RS3*[*rule-format*,*simp*]: *wellformed-policy1-strong p* $\longrightarrow$
*wellformed-policy1-strong* (*removeShadowRules3 Cp p*)
⟨*proof*⟩

**lemma** *AILRS3*[*rule-format*,*simp*]: *all-in-list p l* $\longrightarrow$
*all-in-list* (*removeShadowRules3 Cp p*) *l*
⟨*proof*⟩

**lemma** *SCRS3*[*rule-format*,*simp*]: *singleCombinators p* $\longrightarrow$
*singleCombinators*(*removeShadowRules3 Cp p*)
⟨*proof*⟩

**lemma** *RS3subset*: *set* (*removeShadowRules3 Cp p*) $\subseteq$ *set p*
⟨*proof*⟩

**lemma** *ANDRS3*[*simp*]: ⟦*singleCombinators p*; *allNetsDistinct p*⟧ ⟹
*allNetsDistinct* (*removeShadowRules3 Cp p*)
⟨*proof*⟩

**lemma** *nlpaux*: $x \notin dom\ (Cp\ b) \Longrightarrow Cp\ (a \oplus b)\ x = Cp\ a\ x$
⟨*proof*⟩

**lemma** *notindom*[*rule-format*]: $a \in set\ p \longrightarrow\ x \notin dom\ (Cp\ (list2FWpolicy\ p)) \longrightarrow$
$x \notin dom\ (Cp\ a)$
⟨*proof*⟩

**lemma** *C-eq-rd*[*rule-format*]: $p \neq [] \Longrightarrow$
$Cp\ (list2FWpolicy\ (remdups\ p)) = Cp\ (list2FWpolicy\ p)$
⟨*proof*⟩

**lemma** *nMT-domMT*: ⟦¬ *notMTpolicy Cp p*; $p \neq []$⟧ ⟹ $r \notin dom\ (Cp\ (list2FWpolicy\ p))$
⟨*proof*⟩

**lemma** *C-eq-RS3-aux*[*rule-format*]: *notMTpolicy Cp p* ⟹
$Cp\ (list2FWpolicy\ p)\ x = Cp\ (list2FWpolicy\ (removeShadowRules3\ Cp\ p))\ x$
⟨*proof*⟩

**lemma** *C-eq-id*: *wellformed-policy1-strong p* ⟹
$Cp(list2FWpolicy\ (insertDeny\ p)) = Cp\ (list2FWpolicy\ p)$
⟨*proof*⟩

**lemma** *C-eq-RS3*: *notMTpolicy Cp p* ⟹
$Cp(list2FWpolicy\ (removeShadowRules3\ Cp\ p)) = Cp\ (list2FWpolicy\ p)$
⟨*proof*⟩

**lemma** *NMPrd*[*rule-format*]: *notMTpolicy Cp p* ⟶ *notMTpolicy Cp* (*remdups p*)
⟨*proof*⟩

**lemma** *NMPDA*[*rule-format*]: *DenyAll* ∈ *set p* ⟶ *notMTpolicy Cp p*
⟨*proof*⟩

**lemma** *NMPiD*[*rule-format*]: *notMTpolicy Cp* (*insertDeny p*)
⟨*proof*⟩

**lemma** *list2FWpolicy2list*[*rule-format*]: *Cp* (*list2FWpolicy*(*policy2list p*)) = (*Cp p*)
⟨*proof*⟩

**lemmas** *C-eq-Lemmas* = *noneMTRS2 noneMTrd SCp2l*
*wp1n-RS2 wp1ID NMPiD wp1alternative-RS1*
*p2lNmt list2FWpolicy2list wellformed-policy3-charn waux2 wp1-eq*

**lemmas** *C-eq-subst-Lemmas* = *C-eq-sorted C-eq-sortedQ C-eq-RS2 C-eq-rd C-eq-RS3 C-eq-id*

**lemma** *C-eq-All-untilSorted*:
⟦*DenyAll* ∈ *set* (*policy2list p*); *all-in-list* (*policy2list p*) *l*;
*allNetsDistinct* (*policy2list p*)⟧ ⟹
*Cp*(*list2FWpolicy* (*sort* (*removeShadowRules2* (*remdups* (*removeShadowRules3 Cp*
(*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) *l*)) = *Cp p*
⟨*proof*⟩

**lemma** *C-eq-All-untilSortedQ*:
⟦*DenyAll* ∈ *set* (*policy2list p*); *all-in-list* (*policy2list p*) *l*;
*allNetsDistinct* (*policy2list p*)⟧ ⟹
*Cp*(*list2FWpolicy* (*qsort* (*removeShadowRules2* (*remdups* (*removeShadowRules3 Cp*
(*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) *l*)) = *Cp p*
⟨*proof*⟩

**lemma** *C-eq-All-untilSorted-withSimps*:
⟦*DenyAll* ∈ *set* (*policy2list p*); *all-in-list* (*policy2list p*) *l*;
*allNetsDistinct* (*policy2list p*)⟧ ⟹
*Cp*(*list2FWpolicy* (*sort* (*removeShadowRules2* (*remdups* (*removeShadowRules3 Cp* (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) *l*)) = *Cp p*
⟨*proof*⟩

**lemma** *C-eq-All-untilSorted-withSimpsQ*:
⟦*DenyAll* ∈ *set* (*policy2list p*); *all-in-list* (*policy2list p*) *l*;
*allNetsDistinct* (*policy2list p*)⟧ ⟹
*Cp*(*list2FWpolicy* (*qsort* (*removeShadowRules2* (*remdups* (*removeShadowRules3 Cp* (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) *l*)) = *Cp p*
⟨*proof*⟩

**lemma** *InDomConc*[*rule-format*]: $p \neq [] \longrightarrow x \in dom\ (Cp\ (list2FWpolicy\ (p))) \longrightarrow$
$x \in\ dom\ (Cp\ (list2FWpolicy\ (a\#p)))$
⟨*proof*⟩

**lemma** *not-in-member*[*rule-format*]: $member\ a\ b \longrightarrow x \notin dom\ (Cp\ b) \longrightarrow x \notin dom\ (Cp\ a)$
⟨*proof*⟩

**lemma** *src-in-sdnets*[*rule-format*]: $\neg\ member\ DenyAll\ x \longrightarrow p \in dom\ (Cp\ x) \longrightarrow$
$subnetsOfAdr\ (src\ p) \cap (fst\text{-}set\ (sdnets\ x)) \neq \{\}$
⟨*proof*⟩

**lemma** *dest-in-sdnets*[*rule-format*]: $\neg\ member\ DenyAll\ x \longrightarrow p \in dom\ (Cp\ x) \longrightarrow$
$subnetsOfAdr\ (dest\ p) \cap (snd\text{-}set\ (sdnets\ x)) \neq \{\}$
⟨*proof*⟩

**lemma** *sdnets-in-subnets*[*rule-format*]: $p \in dom\ (Cp\ x) \longrightarrow \neg\ member\ DenyAll\ x \longrightarrow$
$(\exists\ (a,b) \in sdnets\ x.\ a \in subnetsOfAdr\ (src\ p) \wedge b \in subnetsOfAdr\ (dest\ p))$
⟨*proof*⟩

**lemma** *disjSD-no-p-in-both*[*rule-format*]:
$[\![disjSD\text{-}2\ x\ y;\ \neg\ member\ DenyAll\ x;\ \neg\ member\ DenyAll\ y;$
$p \in dom\ (Cp\ x);\ p \in dom\ (Cp\ y)]\!] \Longrightarrow False$
⟨*proof*⟩

**lemma** *list2FWpolicy-eq*: $zs \neq [] \Longrightarrow$
$Cp\ (list2FWpolicy\ (x \oplus y\ \#\ z))\ p = Cp\ (x \oplus list2FWpolicy\ (y\ \#\ z))\ p$
⟨*proof*⟩

**lemma** *dom-sep*[*rule-format*]: $x \in dom\ (Cp\ (list2FWpolicy\ p)) \longrightarrow$
$x \in dom\ (Cp\ (list2FWpolicy(separate\ p)))$
⟨*proof*⟩

**lemma** *domdConcStart*[*rule-format*]: $x \in dom\ (Cp\ (list2FWpolicy\ (a\#b))) \longrightarrow$
$x \notin dom\ (Cp\ (list2FWpolicy\ b))$
$\longrightarrow x \in dom\ (Cp\ (a))$
⟨*proof*⟩

**lemma** *sep-dom2-aux*: $[\![x \in dom\ (Cp\ (list2FWpolicy\ (a \oplus y\ \#\ z)))]\!]$
$\Longrightarrow x \in dom\ (Cp\ (a \oplus list2FWpolicy\ (y\ \#\ z)))$
⟨*proof*⟩

**lemma** *sep-dom2-aux2*:
$[\![(x \in dom\ (Cp\ (list2FWpolicy\ (separate\ (y\ \#\ z)))) \longrightarrow$
$x \in dom\ (Cp\ (list2FWpolicy\ (y\ \#\ z)))); x \in dom\ (Cp\ (list2FWpolicy\ (a\ \#\ separate\ (y\ \#\ z))))]\!]$
$\Longrightarrow x \in dom\ (Cp\ (list2FWpolicy\ (a \oplus y\ \#\ z)))$
⟨*proof*⟩

**lemma** *sep-dom2*[*rule-format*]:
$x \in dom\ (Cp\ (list2FWpolicy\ (separate\ p))) \longrightarrow x \in dom\ (Cp\ (list2FWpolicy(\ p)))$
⟨*proof*⟩

**lemma** *sepDom*: $dom\ (Cp\ (list2FWpolicy\ p)) = dom\ (Cp\ (list2FWpolicy\ (separate\ p)))$
⟨*proof*⟩

**lemma** *C-eq-s-ext*[*rule-format*]: $p \neq [] \longrightarrow$
$Cp\ (list2FWpolicy\ (separate\ p))\ a\ = Cp\ (list2FWpolicy\ p)\ a$
⟨*proof*⟩

**lemma** *C-eq-s*: $p \neq [] \Longrightarrow Cp\ (list2FWpolicy\ (separate\ p)) = Cp\ (list2FWpolicy\ p)$
⟨*proof*⟩

**lemma** *sortnMTQ*: $p \neq [] \Longrightarrow qsort\ p\ l \neq []$
⟨*proof*⟩

**lemmas** *C-eq-Lemmas-sep* =
*C-eq-Lemmas sortnMT sortnMTQ RS2-NMT NMPrd NMPRS3*

**lemma** *C-eq-until-separated*:
$[\![DenyAll \in set\ (policy2list\ p);\ all\text{-}in\text{-}list\ (policy2list\ p)\ l;$
$allNetsDistinct\ (policy2list\ p)]\!] \Longrightarrow$
$Cp\ (list2FWpolicy\ (separate\ (sort\ (removeShadowRules2\ (remdups\ (removeShadowRules3\ Cp$
$(insertDeny\ (removeShadowRules1\ (policy2list\ p))))))\ l))) = Cp\ p$
⟨*proof*⟩

**lemma** *C-eq-until-separatedQ*:
$[\![DenyAll \in set\ (policy2list\ p);\ all\text{-}in\text{-}list\ (policy2list\ p)\ l;$
$allNetsDistinct\ (policy2list\ p)]\!] \Longrightarrow$
$Cp\ (list2FWpolicy\ (separate\ (qsort\ (removeShadowRules2\ (remdups\ (removeShadowRules3\ Cp$
$(insertDeny\ (removeShadowRules1\ (policy2list\ p))))))\ l))) = Cp\ p$
⟨*proof*⟩

**lemma** *domID*[*rule-format*]: $p \neq [] \wedge x \in dom(Cp(list2FWpolicy\ p)) \longrightarrow$
$x \in dom\ (Cp(list2FWpolicy(insertDenies\ p)))$

⟨*proof*⟩

**lemma** *DA-is-deny*:
$x \in dom\ (Cp\ (DenyAllFromTo\ a\ b \oplus DenyAllFromTo\ b\ a \oplus DenyAllFromTo\ a\ b)) \Longrightarrow$
$Cp\ (DenyAllFromTo\ a\ b \oplus DenyAllFromTo\ b\ a \oplus DenyAllFromTo\ a\ b)\ x = Some\ (deny\ ())$
⟨*proof*⟩

**lemma** *iDdomAux*[*rule-format*]:
$p \neq [] \longrightarrow x \notin dom\ (Cp\ (list2FWpolicy\ p)) \longrightarrow$
$x \in dom\ (Cp\ (list2FWpolicy\ (insertDenies\ p))) \longrightarrow$
$Cp\ (list2FWpolicy\ (insertDenies\ p))\ x = Some\ (deny\ ())$
⟨*proof*⟩

**lemma** *iD-isD*[*rule-format*]: $p \neq [] \longrightarrow x \notin dom\ (Cp\ (list2FWpolicy\ p))$
$\longrightarrow Cp\ (DenyAll \oplus list2FWpolicy\ (insertDenies\ p))\ x = Cp\ DenyAll\ x$
⟨*proof*⟩

**lemma** *inDomConc*:$[\![ x \notin dom\ (Cp\ a);\ x \notin dom\ (Cp\ (list2FWpolicy\ p))]\!] \Longrightarrow$
$x \notin dom\ (Cp\ (list2FWpolicy(a\#p)))$
⟨*proof*⟩

**lemma** *domsdisj*[*rule-format*]: $p \neq [] \longrightarrow (\forall\ x\ s.\ s \in set\ p \wedge x \in dom\ (Cp\ A) \longrightarrow$
$x \notin dom\ (Cp\ s)) \longrightarrow y \in dom\ (Cp\ A) \longrightarrow$
$y \notin dom\ (Cp\ (list2FWpolicy\ p))$
⟨*proof*⟩

**lemma** *isSepaux*:
$[\![p \neq [];\ noDenyAll\ (a\#p);\ separated\ (a\ \#\ p);$
$x \in dom\ (Cp\ (DenyAllFromTo\ (first\text{-}srcNet\ a)\ (first\text{-}destNet\ a) \oplus$
$DenyAllFromTo\ (first\text{-}destNet\ a)\ (first\text{-}srcNet\ a) \oplus a))]\!] \Longrightarrow$
$x \notin dom\ (Cp\ (list2FWpolicy\ p))$
⟨*proof*⟩

**lemma** *noneMTsep*[*rule-format*]: $noneMT\ Cp\ p \longrightarrow noneMT\ Cp\ (separate\ p)$
⟨*proof*⟩

**lemma** *dom-id*:
$[\![noDenyAll\ (a\#p);\ separated\ (a\#p);\ p \neq [];\ x \notin dom\ (Cp\ (list2FWpolicy\ p));$
$x \in dom\ (Cp\ (a))]\!]$
$\Longrightarrow x \notin dom\ (Cp\ (list2FWpolicy\ (insertDenies\ p)))$
⟨*proof*⟩

**lemma** *C-eq-iD-aux2*[*rule-format*]:

*noDenyAll1 p* $\longrightarrow$
*separated p* $\longrightarrow$
$p \neq []$ $\longrightarrow$
$x \in dom\ (Cp\ (list2FWpolicy\ p))$ $\longrightarrow$
$Cp(list2FWpolicy\ (insertDenies\ p))\ x = Cp(list2FWpolicy\ p)\ x$
⟨*proof*⟩

**lemma** *C-eq-iD*: $[\![separated\ p;\ noDenyAll1\ p;\ wellformed\text{-}policy1\text{-}strong\ p]\!] \Longrightarrow$
$Cp\ (list2FWpolicy\ (insertDenies\ p)) = Cp\ (list2FWpolicy\ p)$
⟨*proof*⟩

**lemma** *noDAsortQ*[*rule-format*]: *noDenyAll1 p* $\longrightarrow$ *noDenyAll1* $(qsort\ p\ l)$
⟨*proof*⟩

**lemma** *NetsCollectedSortQ*: *distinct p* $\Longrightarrow$ *noDenyAll1 p* $\Longrightarrow$ *all-in-list p l* $\Longrightarrow$
*singleCombinators p* $\Longrightarrow$ *NetsCollected* $(qsort\ p\ l)$
⟨*proof*⟩

**lemmas** *CLemmas* = *nMTSort nMTSortQ noneMTRS2 noneMTrd*
*noDAsort noDAsortQ nDASC wp1-eq wp1ID*
*SCp2l ANDSep wp1n-RS2*
*OTNSEp OTNSC noDA1sep wp1-alternativesep wellformed-eq*
*wellformed1-alternative-sorted*

**lemmas** *C-eqLemmas-id* = *CLemmas NC2Sep NetsCollectedSep*
*NetsCollectedSort NetsCollectedSortQ separatedNC*

**lemma** *C-eq-Until-InsertDenies*: $[\![DenyAll \in set\ (policy2list\ p);\ all\text{-}in\text{-}list$
$(policy2list\ p)\ l;\ allNetsDistinct\ (policy2list\ p)]\!] \Longrightarrow$
$Cp\ (list2FWpolicy\ ((insertDenies\ (separate\ (sort\ (removeShadowRules2\ (remdups$
$(removeShadowRules3\ Cp\ (insertDeny\ (removeShadowRules1\ (policy2list\ p))))))\ l))))) =$
$Cp\ p$
⟨*proof*⟩

**lemma** *C-eq-Until-InsertDeniesQ*: $[\![DenyAll \in set\ (policy2list\ p);\ all\text{-}in\text{-}list$
$(policy2list\ p)\ l;\ allNetsDistinct\ (policy2list\ p)]\!] \Longrightarrow$
$Cp\ (list2FWpolicy\ ((insertDenies\ (separate\ (qsort\ (removeShadowRules2\ (remdups$
$(removeShadowRules3\ Cp\ (insertDeny\ (removeShadowRules1\ (policy2list\ p))))))\ l))))) =$
$Cp\ p$
⟨*proof*⟩

**lemma** *C-eq-RD-aux*[*rule-format*]: *Cp* (*p*) *x* = *Cp* (*removeDuplicates p*) *x*
⟨*proof*⟩

**lemma** *C-eq-RAD-aux*[*rule-format*]:
*p* ≠ [] ⟶ *Cp* (*list2FWpolicy p*) *x* = *Cp* (*list2FWpolicy* (*removeAllDuplicates p*)) *x*
⟨*proof*⟩

**lemma** *C-eq-RAD*:
*p* ≠ [] ⟹ *Cp* (*list2FWpolicy p*) = *Cp* (*list2FWpolicy* (*removeAllDuplicates p*))
⟨*proof*⟩

**lemma** *C-eq-compile*:
⟦*DenyAll* ∈ *set* (*policy2list p*); *all-in-list* (*policy2list p*) *l*;
*allNetsDistinct* (*policy2list p*)⟧ ⟹
*Cp* (*list2FWpolicy* (*removeAllDuplicates* (*insertDenies* (*separate* (*sort*
(*removeShadowRules2* (*remdups* (*removeShadowRules3 Cp* (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) *l*))))) = *Cp p*
⟨*proof*⟩

**lemma** *C-eq-compileQ*:
⟦*DenyAll* ∈ *set* (*policy2list p*); *all-in-list* (*policy2list p*) *l*;
*allNetsDistinct* (*policy2list p*)⟧ ⟹
*Cp* (*list2FWpolicy* (*removeAllDuplicates* (*insertDenies* (*separate* (*qsort*
(*removeShadowRules2* (*remdups* (*removeShadowRules3 Cp* (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) *l*))))) = *Cp p*
⟨*proof*⟩

**lemma** *C-eq-normalizePr*:
⟦*DenyAll* ∈ *set* (*policy2list p*);
*allNetsDistinct* (*policy2list p*);
*all-in-list* (*policy2list p*) (*Nets-List p*)⟧ ⟹
*Cp* (*list2FWpolicy* (*normalizePr p*)) = *Cp p*
⟨*proof*⟩

**lemma** *C-eq-normalizePrQ*:
⟦*DenyAll* ∈ *set* (*policy2list p*);
*allNetsDistinct* (*policy2list p*);
*all-in-list* (*policy2list p*) (*Nets-List p*)⟧ ⟹
*Cp* (*list2FWpolicy* (*normalizePrQ p*)) = *Cp p*
⟨*proof*⟩

**lemma** *domSubset3*: $dom\ (Cp\ (DenyAll \oplus x)) = dom\ (Cp\ (DenyAll))$
⟨*proof*⟩

**lemma** *domSubset4*: $dom\ (Cp\ (DenyAllFromTo\ x\ y \oplus DenyAllFromTo\ y\ x \oplus AllowPortFromTo\ x\ y\ dn)) =$
$dom\ (Cp\ (DenyAllFromTo\ x\ y \oplus DenyAllFromTo\ y\ x))$
⟨*proof*⟩

**lemma** *domSubset5*:
$dom\ (Cp\ (DenyAllFromTo\ x\ y \oplus DenyAllFromTo\ y\ x \oplus AllowPortFromTo\ y\ x\ dn)) =$
$dom\ (Cp\ (DenyAllFromTo\ x\ y \oplus DenyAllFromTo\ y\ x))$
⟨*proof*⟩

**lemma** *domSubset1*: $dom\ (Cp\ (DenyAllFromTo\ one\ two \oplus DenyAllFromTo\ two\ one \oplus AllowPortFromTo\ one\ two\ dn \oplus x)) =$
$dom\ (Cp\ (DenyAllFromTo\ one\ two \oplus DenyAllFromTo\ two\ one \oplus x))$
⟨*proof*⟩

**lemma** *domSubset2*:
$dom\ (Cp\ (DenyAllFromTo\ one\ two \oplus DenyAllFromTo\ two\ one \oplus AllowPortFromTo\ two\ one\ dn \oplus x)) =$
$dom\ (Cp\ (DenyAllFromTo\ one\ two \oplus DenyAllFromTo\ two\ one \oplus x))$
⟨*proof*⟩

**lemma** *ConcAssoc2*: $Cp\ (X \oplus Y \oplus ((A \oplus B) \oplus D)) = Cp\ (X \oplus Y \oplus A \oplus B \oplus D)$
⟨*proof*⟩

**lemma** *ConcAssoc3*: $Cp\ (X \oplus ((Y \oplus A) \oplus D)) = Cp\ (X \oplus Y \oplus A \oplus D)$
⟨*proof*⟩

**lemma** *RS3-NMT*[*rule-format*]: $DenyAll \in set\ p \longrightarrow$
$removeShadowRules3\ Cp\ p \neq []$
⟨*proof*⟩

**lemma** *norm-notMT*: $DenyAll \in set\ (policy2list\ p) \Longrightarrow normalizePr\ p \neq []$
⟨*proof*⟩

**lemma** *norm-notMTQ*: $DenyAll \in set\ (policy2list\ p) \Longrightarrow normalizePrQ\ p \neq []$
⟨*proof*⟩

**lemma** *domDA*: $dom\ (Cp\ (DenyAll \oplus A)) = dom\ (Cp\ (DenyAll))$
⟨*proof*⟩

**lemmas** *domain-reasoningPr = domDA ConcAssoc2 domSubset1 domSubset2 domSubset3 domSubset4 domSubset5 domSubsetDistr1 domSubsetDistr2 domSubsetDistrA domSubsetDistrD coerc-assoc ConcAssoc ConcAssoc3*

The following lemmas help with the normalisation

**lemma** *list2policyR-Start*[*rule-format*]: $p \in dom\ (Cp\ a) \longrightarrow Cp\ (list2policyR\ (a\ \#\ list))\ p = Cp\ a\ p$
⟨*proof*⟩

**lemma** *list2policyR-End*: $p \notin dom\ (Cp\ a) \Longrightarrow Cp\ (list2policyR\ (a\ \#\ list))\ p = (Cp\ a \bigoplus list2policy\ (map\ Cp\ list))\ p$
⟨*proof*⟩

**lemma** *l2polR-eq-el*[*rule-format*]: $N \neq [] \longrightarrow Cp(\ list2policyR\ N)\ p = (list2policy\ (map\ Cp\ N))\ p$
⟨*proof*⟩

**lemma** *l2polR-eq*: $N \neq [] \Longrightarrow Cp(\ list2policyR\ N) = (list2policy\ (map\ Cp\ N))$
⟨*proof*⟩

**lemma** *list2FWpolicys-eq-el*[*rule-format*]:
$Filter \neq [] \longrightarrow Cp\ (list2policyR\ Filter)\ p = Cp\ (list2FWpolicy\ (rev\ Filter))\ p$
⟨*proof*⟩

**lemma** *list2FWpolicys-eq*:
$Filter \neq [] \Longrightarrow Cp\ (list2policyR\ Filter) = Cp\ (list2FWpolicy\ (rev\ Filter))$
⟨*proof*⟩

**lemma** *list2FWpolicys-eq-sym*:
$Filter \neq [] \Longrightarrow$

$Cp\ (list2policyR\ (rev\ Filter)) = \ Cp\ (list2FWpolicy\ Filter)$
⟨*proof*⟩

**lemma** *p-eq*[*rule-format*]: $p \neq [] \longrightarrow$
$list2policy\ (map\ Cp\ (rev\ p)) = Cp\ (list2FWpolicy\ p)$
⟨*proof*⟩

**lemma** *p-eq2*[*rule-format*]: $normalizePr\ x \neq [] \longrightarrow$
$Cp\ (list2FWpolicy\ (normalizePr\ x)) = Cp\ x \longrightarrow$
$list2policy\ (map\ Cp\ (rev\ (normalizePr\ x))) = Cp\ x$
⟨*proof*⟩

**lemma** *p-eq2Q*[*rule-format*]: $normalizePrQ\ x \neq [] \longrightarrow$
$Cp\ (list2FWpolicy\ (normalizePrQ\ x)) = Cp\ x \longrightarrow$
$list2policy\ (map\ Cp\ (rev\ (normalizePrQ\ x))) = Cp\ x$
⟨*proof*⟩

**lemma** *list2listNMT*[*rule-format*]: $x \neq [] \longrightarrow map\ sem\ x \neq []$
⟨*proof*⟩

**lemma** *Norm-Distr2*:
$r\ o\text{-}f\ ((P \bigotimes_2 (list2policy\ Q))\ o\ d) =$
$(list2policy\ ((P \bigotimes_L Q)\ (op \bigotimes_2)\ r\ d))$
⟨*proof*⟩

**lemma** *NATDistr*: $[\![N \neq []; F = Cp\ (list2policyR\ N)]\!] \Longrightarrow$
$((\lambda\ (x,y).\ x)\ o\text{-}f\ ((NAT \bigotimes_2 F)\ o\ (\lambda\ x.\ (x,x))) =$
$(list2policy\ (\ ((NAT \bigotimes_L (map\ Cp\ N))\ (op \bigotimes_2)$
$(\lambda\ (x,y).\ x)\ (\lambda\ x.\ (x,x))))))$
⟨*proof*⟩

**lemma** *C-eq-normalize-manual*:
$[\![DenyAll \in set\ (policy2list\ p);$
$allNetsDistinct\ (policy2list\ p);$
$all\text{-}in\text{-}list\ (policy2list\ p)\ l]\!] \Longrightarrow$
$Cp\ (list2FWpolicy\ (normalize\text{-}manual\text{-}orderPr\ p\ l)) = Cp\ p$
⟨*proof*⟩

**lemma** *p-eq2-manualQ*[*rule-format*]: *normalize-manual-orderPrQ x l* $\neq$ [] $\longrightarrow$
*Cp* (*list2FWpolicy* (*normalize-manual-orderPrQ x l*)) = *Cp x* $\longrightarrow$
*list2policy* (*map Cp* (*rev* (*normalize-manual-orderPrQ x l*))) = *Cp x*
⟨*proof*⟩

**lemma** *norm-notMT-manualQ*: *DenyAll* $\in$ *set* (*policy2list p*) $\Longrightarrow$ *normalize-manual-orderPrQ p l* $\neq$ []
⟨*proof*⟩

**lemma** *C-eq-normalizePr-manualQ*:
⟦*DenyAll* $\in$ *set* (*policy2list p*);
*allNetsDistinct* (*policy2list p*);
*all-in-list* (*policy2list p*) *l*⟧ $\Longrightarrow$
*Cp* (*list2FWpolicy* (*normalize-manual-orderPrQ p l*)) = *Cp p*
⟨*proof*⟩

**lemma** *p-eq2-manual*[*rule-format*]: *normalize-manual-orderPr x l* $\neq$ [] $\longrightarrow$
*Cp* (*list2FWpolicy* (*normalize-manual-orderPr x l*)) = *Cp x* $\longrightarrow$
*list2policy* (*map Cp* (*rev* (*normalize-manual-orderPr x l*))) = *Cp x*
⟨*proof*⟩

**lemma** *norm-notMT-manual*: *DenyAll* $\in$ *set* (*policy2list p*) $\Longrightarrow$ *normalize-manual-orderPr p l* $\neq$ []
⟨*proof*⟩

As an example, how this theorems can be used for a concrete normalisation instantiation.

**lemma** *normalizePrNAT*: ⟦*DenyAll* $\in$ *set* (*policy2list Filter*);
*allNetsDistinct* (*policy2list Filter*);
*all-in-list* (*policy2list Filter*) (*Nets-List Filter*)⟧ $\Longrightarrow$
((λ (*x*,*y*). *x*) *o-f* (((*NAT* $\bigotimes_2$ *Cp Filter*) *o* (λ*x*. (*x*,*x*))))) =
*list2policy* ((*NAT* $\bigotimes_L$ (*map Cp* (*rev* (*normalizePr Filter*)))) (*op* $\bigotimes_2$) (λ (*x*,*y*). *x*) (λ *x*. (*x*,*x*)))
⟨*proof*⟩

**lemma** *domSimpl*[*simp*]: *dom* (*Cp* (*A* $\oplus$ *DenyAll*)) = *dom* (*Cp* (*DenyAll*))
⟨*proof*⟩

**end**

## References


[1] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. Verified firewall policy transformations for test case generation. In A. Cavalli and S. Ghosh, editors, *International Conference on Software Testing (ICST10)*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

[2] A. D. Brucker, L. Brügger, and B. Wolff. Model-based firewall conformance testing. In K. Suzuki and T. Higashino, editors, *Testcom/FATES 2008*, number 5047 in Lecture Notes in Computer Science, pages 103–118. Springer-Verlag, 2008.

[3] A. D. Brucker and B. Wolff. Test-sequence generation with HOL-TestGen – with an application to firewall testing. In B. Meyer and Y. Gurevich, editors, *TAP 2007: Tests And Proofs*, number 4454 in Lecture Notes in Computer Science, pages 149–168. Springer-Verlag, 2007.

[4] D. von Bidder. *Specification-based Firewall Testing.* Ph.d. thesis, ETH Zurich, 2007. ETH Dissertation No. 17172. Diana von Bidder's maiden name is Diana Senn.