

Iptables-Semantics

Cornelius Diekmann, Lars Hupel

March 17, 2025

Abstract

We present a big step semantics of the filtering behavior of the Linux/netfilter iptables firewall. We provide algorithms to simplify complex iptables rule sets to a simple firewall model (c.f. AFP entry `Simple_Firewall`) and to verify spoofing protection of a rule set. Internally, we embed our semantics into ternary logic, ultimately supporting every iptables match condition by abstracting over unknowns. Using this AFP entry and all entries it depends on, we created an easy-to-use, stand-alone haskell tool called *ffwu* (<http://iptables.isabelle.systems>). The tool does not require any input —except for the `iptables-save` dump of the analyzed firewall— and presents interesting results about the user’s rule set. Real-World firewall errors have been uncovered, as well as the correctness of rule sets has been proven with the help of our tool.

For a detailed description, see [2, 4, 3, 1].

Acknowledgements This entry would not have been possible without the help of Julius Michaelis, Max Haslbeck, Stephan-A. Posselt, Lars Noschinski, Manuel Eberl, Gerwin Klein, the Isabelle group Munich, and Georg Carle.

Contents

1	Repeat finitely Until it Stabilizes	5
2	Firewall Basic Syntax	6
3	Basic Algorithms	6
4	Big Step Semantics	10
4.1	Boolean Matcher Algebra	20
4.2	Add match	21
4.3	Background Ruleset Updating	23

5	Call Return Unfolding	25
5.1	Completeness	26
5.2	<i>process-ret</i> correctness	27
5.3	Soundness	28
6	Ternary Logic	30
6.1	Negation Normal Form	34
7	Packet Matching in Ternary Logic	35
7.1	Ternary Matcher Algebra	38
7.2	Removing Unknown Primitives	39
8	Embedded Ternary-Matching Big Step Semantics	43
8.1	Ternary Semantics (Big Step)	43
8.2	wf ruleset	45
8.3	Ternary Semantics (Function)	46
8.3.1	Append, Prepend, Postpend, Composition	47
8.4	Equality with $\gamma, p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t$ semantics	48
8.5	Matching	50
9	IPv4 Addresses	52
9.1	IPv4 Addresses in IPTables Notation (how we parse it)	52
10	Matching TCP Flags	55
11	Ports (layer 4)	57
12	Tagged Simple Packet	60
13	Primitive Matchers: Interfaces, IP Space, Layer 4 Ports Matcher	61
14	Approximate Matching Tactics	62
14.1	A Generic primitive matcher: Agnostic of IP Addresses	63
14.2	Basic optimisations	65
14.3	Primitive Matchers: IP Port Iface Matcher	66
14.4	Basic optimisations	69
14.5	Abstracting over unknowns	70
15	Examples Big Step Semantics	71
16	Semantics Stateful	77
16.1	Model 1 – Curried Stateful Matcher	77
16.2	Model 2 – Packets Tagged with State Information	78
16.3	Example: Conntrack with curried matcher	79
16.4	Example: Conntrack with packet tagging	81

17 Big Step Semantics with Goto	82
17.1 Semantics	82
17.1.1 Forward reasoning	85
17.2 Determinism	88
17.3 Matching	88
17.4 Goto Unfolding	89
18 Negation Type DNF	92
18.0.1 inverting a DNF	93
18.0.2 Optimizing	94
19 Boolean Matching vs. Ternary Matching	94
20 Fixed Action	96
20.1 <i>match-list</i>	97
21 Normalized (DNF) matches	100
22 Normalizing rules instead of only match expressions	101
22.1 Functions which preserve <i>normalized-nnf-match</i>	104
23 Negation Type Matching	104
24 Primitive Normalization	107
24.1 Normalized Primitives	107
24.2 Primitive Extractor	110
24.3 Normalizing and Optimizing Primitives	113
24.4 Optimizing a match expression	116
24.5 Processing a list of normalization functions	118
25 Combine Match Expressions	119
26 Further Lemmas about the Common Matcher	120
27 Normalizing L4 Ports	121
27.1 Defining Normalized Ports	121
27.2 Compressing Positive Matches on Ports into a Single Match .	122
27.3 Rewriting Negated Matches on Ports	123
27.4 Normalizing Positive Matches on Ports	127
27.5 Complete Normalization	130
27.6 Normalizing IP Addresses	139
27.7 Optimizing interfaces in match expressions	141
28 Word Upto	145
29 Optimizing Protocols	147

30 Optimizing protocols in match expressions	147
30.1 Importing the matches on <i>primitive-protocol</i> from <i>L4Ports</i> . . .	150
30.2 Putting things together	152
31 Reverse Remdups	153
31.1 Sanity checking for an <i>'i ipassignment</i>	154
31.2 IP Assignment difference	159
32 No Spoofing	161
32.1 Spoofing Protection	161
33 Firewall toString Functions	170
34 Routing and IP Assignments	172
34.1 Routing IP Assignment	173
35 Replacing output interfaces by their IP ranges according to Routing	174
36 Trying to connect inbound interfaces by their IP ranges	176
36.1 Constraining Interfaces	176
36.2 Sanity checking the assumption	177
36.3 Replacing Interfaces Completely	178
37 Optimizing	179
37.1 Removing Shadowed Rules	179
37.1.1 Soundness	180
37.2 Removing rules which cannot apply	180
38 Optimizing and Normalizing Primitives	181
39 Transforming rulesets	183
39.1 Optimizations	183
39.2 Optimize and Normalize to NNF form	183
39.3 Abstracting over unknowns	184
39.4 Normalizing and Transforming Primitives	185
40 Abstracting over Primitives	194
41 Iptables to Simple Firewall and Vice Versa	198
41.1 Simple Match to MatchExpr	198
41.2 MatchExpr to Simple Match	199
41.2.1 Normalizing Interfaces	200
41.2.2 Normalizing Protocols	200

42 Semantics Embedding	205
42.1 Tactic <i>in-doubt-allow</i>	205
42.2 Tactic <i>in-doubt-deny</i>	206
42.3 Approximating Closures	207
42.4 Exact Embedding	208
43 Normalizing Rulesets in the Boolean Big Step Semantics	209
44 Code Interface	209
44.1 L4 Ports Parser Helper	211
45 Parser for iptables-save	213
46 An SML Parser for iptables-save	213
47 Spoofing protection in Ternary Semantics implies Spoofing protection Boolean Semantics	214
48 Parser for iptables-save	215
49 An SML Parser for iptables-save	215
50 Applying the Access Matrix to the Bigstep Semantics	217
51 Documentation	219
51.1 General Model	219
51.2 Unfolding the Ruleset	220
51.3 Spoofing protection	221
51.4 Simple Firewall Model	221
51.5 Service Matrices	222

1 Repeat finitely Until it Stabilizes

```
theory Repeat-Stabilize
imports Main
begin
```

Repeating something a number of times

Iterating a function at most n times (first parameter) until it stabilizes.

```
fun repeat-stabilize :: nat => ('a => 'a) => 'a => 'a where
  repeat-stabilize 0 v = v |
  repeat-stabilize (Suc n) f v = (let v-new = f v in if v = v-new then v else re-
    peat-stabilize n f v-new)
```

```
lemma repeat-stabilize-funpow: repeat-stabilize n f v = (f~n) v
  <proof>
```

lemma *repeat-stabilize-induct*: $(P\ m) \implies (\bigwedge m. P\ m \implies P\ (f\ m)) \implies P\ (\text{repeat-stabilize}\ n\ f\ m)$
 ⟨*proof*⟩

end

2 Firewall Basic Syntax

theory *Firewall-Common*
imports *Main Simple-Firewall.Firewall-Common-Decision-State*
Common/Repeat-Stabilize
begin

Our firewall model supports the following actions.

datatype *action* = *Accept* | *Drop* | *Log* | *Reject* | *Call string* | *Return* | *Goto string*
 | *Empty* | *Unknown*

We support the following algebra over primitives of type *'a*. The type parameter *'a* denotes the primitive match condition. For example, matching on source IP address or on protocol. We lift the primitives to an algebra. Note that we do not have an Or expression.

datatype *'a match-expr* = *Match 'a*
 | *MatchNot 'a match-expr*
 | *MatchAnd 'a match-expr 'a match-expr*
 | *MatchAny*

definition *MatchOr* :: *'a match-expr* \Rightarrow *'a match-expr* \Rightarrow *'a match-expr* **where**
MatchOr m1 m2 = *MatchNot (MatchAnd (MatchNot m1) (MatchNot m2))*

A firewall rule consists of a match expression and an action.

datatype *'a rule* = *Rule (get-match: 'a match-expr) (get-action: action)*

lemma *rules-singleton-rew-E*:
 $[Rule\ m\ a] = rs_1 @ rs_2 \implies$
 $(rs_1 = [Rule\ m\ a] \implies rs_2 = [] \implies P\ m\ a) \implies$
 $(rs_1 = [] \implies rs_2 = [Rule\ m\ a] \implies P\ m\ a) \implies P\ m\ a$
 ⟨*proof*⟩

3 Basic Algorithms

These algorithms should be valid for all firewall semantics. The corresponding proofs follow once the semantics are defined.

The actions *Log* and *Empty* do not modify the packet processing in any way. They can be removed.

fun *rm-LogEmpty* :: 'a rule list \Rightarrow 'a rule list **where**
rm-LogEmpty [] = [] |
rm-LogEmpty ((Rule - Empty)#rs) = *rm-LogEmpty* rs |
rm-LogEmpty ((Rule - Log)#rs) = *rm-LogEmpty* rs |
rm-LogEmpty (r#rs) = r # *rm-LogEmpty* rs

lemma *rm-LogEmpty-filter*: *rm-LogEmpty* rs = filter ($\lambda r. \text{get-action } r \neq \text{Log} \wedge \text{get-action } r \neq \text{Empty}$) rs
 <proof>

lemma *rm-LogEmpty-seq*: *rm-LogEmpty* (rs1@rs2) = *rm-LogEmpty* rs1 @ *rm-LogEmpty* rs2
 <proof>

Optimize away MatchAny matches

fun *opt-MatchAny-match-expr-once* :: 'a match-expr \Rightarrow 'a match-expr **where**
opt-MatchAny-match-expr-once MatchAny = MatchAny |
opt-MatchAny-match-expr-once (Match a) = (Match a) |
opt-MatchAny-match-expr-once (MatchNot (MatchNot m)) = (*opt-MatchAny-match-expr-once* m) |
opt-MatchAny-match-expr-once (MatchNot m) = MatchNot (*opt-MatchAny-match-expr-once* m) |
opt-MatchAny-match-expr-once (MatchAnd MatchAny MatchAny) = MatchAny
 |
opt-MatchAny-match-expr-once (MatchAnd MatchAny m) = (*opt-MatchAny-match-expr-once* m) |

opt-MatchAny-match-expr-once (MatchAnd m MatchAny) = (*opt-MatchAny-match-expr-once* m) |
opt-MatchAny-match-expr-once (MatchAnd - (MatchNot MatchAny)) = (MatchNot MatchAny) |
opt-MatchAny-match-expr-once (MatchAnd (MatchNot MatchAny) -) = (MatchNot MatchAny) |
opt-MatchAny-match-expr-once (MatchAnd m1 m2) = MatchAnd (*opt-MatchAny-match-expr-once* m1) (*opt-MatchAny-match-expr-once* m2)

It is still a good idea to apply *opt-MatchAny-match-expr-once* multiple times.

Example:

lemma *MatchNot* (*opt-MatchAny-match-expr-once* (MatchAnd MatchAny (MatchNot MatchAny))) = *MatchNot* (MatchNot MatchAny) <proof>

lemma m = (MatchAnd (MatchAnd MatchAny MatchAny) (MatchAnd MatchAny MatchAny)) \implies
 (*opt-MatchAny-match-expr-once* $\sim\sim$ 2) m \neq *opt-MatchAny-match-expr-once* m <proof>

definition *opt-MatchAny-match-expr* :: 'a match-expr \Rightarrow 'a match-expr **where**
opt-MatchAny-match-expr m \equiv repeat-stabilize 2 *opt-MatchAny-match-expr-once* m

Rewrite *Reject* actions to *Drop* actions. If we just care about the filtering

decision (*FinalAllow* or *FinalDeny*), they should be equal.

fun *rw-Reject* :: 'a rule list \Rightarrow 'a rule list **where**
rw-Reject [] = [] |
rw-Reject ((Rule m Reject)#rs) = (Rule m Drop)#*rw-Reject* rs |
rw-Reject (r#rs) = r # *rw-Reject* rs

We call a ruleset simple iff the only actions are *Accept* and *Drop*

definition *simple-ruleset* :: 'a rule list \Rightarrow bool **where**
simple-ruleset rs $\equiv \forall r \in$ set rs. *get-action* r = *Accept* \vee *get-action* r = *Drop*

lemma *simple-ruleset-tail*: *simple-ruleset* (r#rs) \implies *simple-ruleset* rs \langle proof \rangle

lemma *simple-ruleset-append*: *simple-ruleset* (rs₁ @ rs₂) \longleftrightarrow *simple-ruleset* rs₁
 \wedge *simple-ruleset* rs₂
 \langle proof \rangle

Structural properties about match expressions

fun *has-primitive* :: 'a match-expr \Rightarrow bool **where**
has-primitive MatchAny = False |
has-primitive (Match a) = True |
has-primitive (MatchNot m) = *has-primitive* m |
has-primitive (MatchAnd m1 m2) = (*has-primitive* m1 \vee *has-primitive* m2)

Is a match expression equal to the *MatchAny* expression? Only applicable if no primitives are in the expression.

fun *matcheq-matchAny* :: 'a match-expr \Rightarrow bool **where**
matcheq-matchAny MatchAny \longleftrightarrow True |
matcheq-matchAny (MatchNot m) $\longleftrightarrow \neg$ (*matcheq-matchAny* m) |
matcheq-matchAny (MatchAnd m1 m2) \longleftrightarrow *matcheq-matchAny* m1 \wedge *matcheq-matchAny* m2 |
matcheq-matchAny (Match -) = undefined

fun *matcheq-matchNone* :: 'a match-expr \Rightarrow bool **where**
matcheq-matchNone MatchAny = False |
matcheq-matchNone (Match -) = False |
matcheq-matchNone (MatchNot MatchAny) = True |
matcheq-matchNone (MatchNot (Match -)) = False |
matcheq-matchNone (MatchNot (MatchNot m)) = *matcheq-matchNone* m |
matcheq-matchNone (MatchNot (MatchAnd m1 m2)) \longleftrightarrow *matcheq-matchNone*
(MatchNot m1) \wedge *matcheq-matchNone* (MatchNot m2) |
matcheq-matchNone (MatchAnd m1 m2) \longleftrightarrow *matcheq-matchNone* m1 \vee
matcheq-matchNone m2

lemma *matachAny-matchNone*: \neg *has-primitive* m \implies *matcheq-matchAny* m
 $\longleftrightarrow \neg$ *matcheq-matchNone* m
 \langle proof \rangle

lemma *matcheq-matchNone-no-primitive*: \neg *has-primitive* m \implies *matcheq-matchNone*
(MatchNot m) $\longleftrightarrow \neg$ *matcheq-matchNone* m

$\langle \text{proof} \rangle$

optimizing match expressions

fun *optimize-matches-option* :: ('a match-expr \Rightarrow 'a match-expr option) \Rightarrow 'a rule list \Rightarrow 'a rule list **where**
 optimize-matches-option - [] = [] |
 optimize-matches-option f (Rule m a#rs) = (case f m of None \Rightarrow *optimize-matches-option* f rs | Some m \Rightarrow (Rule m a)#*optimize-matches-option* f rs)

lemma *optimize-matches-option-simple-ruleset*: simple-ruleset rs \Longrightarrow simple-ruleset (*optimize-matches-option* f rs)
 $\langle \text{proof} \rangle$

lemma *optimize-matches-option-preserves*:
($\bigwedge r m. r \in \text{set } rs \Longrightarrow f (\text{get-match } r) = \text{Some } m \Longrightarrow P m$) \Longrightarrow
 $\forall r \in \text{set } (\text{optimize-matches-option } f rs). P (\text{get-match } r)$
 $\langle \text{proof} \rangle$

lemma *optimize-matches-option-append*: *optimize-matches-option* f (rs1@rs2) =
optimize-matches-option f rs1 @ *optimize-matches-option* f rs2
 $\langle \text{proof} \rangle$

definition *optimize-matches* :: ('a match-expr \Rightarrow 'a match-expr) \Rightarrow 'a rule list \Rightarrow 'a rule list **where**
 optimize-matches f rs = *optimize-matches-option* ($\lambda m. (\text{if } \text{matcheq-matchNone } (f m) \text{ then None else Some } (f m))$) rs

lemma *optimize-matches-append*: *optimize-matches* f (rs1@rs2) = *optimize-matches* f rs1 @ *optimize-matches* f rs2
 $\langle \text{proof} \rangle$

lemma *optimize-matches-fst*: *optimize-matches* f (r#rs) = *optimize-matches* f [r]@*optimize-matches* f rs
 $\langle \text{proof} \rangle$

lemma *optimize-matches-preserves*: ($\bigwedge r. r \in \text{set } rs \Longrightarrow P (f (\text{get-match } r))$) \Longrightarrow
 $\forall r \in \text{set } (\text{optimize-matches } f rs). P (\text{get-match } r)$
 $\langle \text{proof} \rangle$

lemma *optimize-matches-simple-ruleset*: simple-ruleset rs \Longrightarrow simple-ruleset (*optimize-matches* f rs)
 $\langle \text{proof} \rangle$

definition *optimize-matches-a* :: (action \Rightarrow 'a match-expr \Rightarrow 'a match-expr) \Rightarrow 'a rule list \Rightarrow 'a rule list **where**

$optimize_matches_a\ f\ rs = map\ (\lambda r. Rule\ (f\ (get_action\ r)\ (get_match\ r))\ (get_action\ r))\ rs$

lemma *optimize-matches-a-simple-ruleset*: $simple_ruleset\ rs \implies simple_ruleset\ (optimize_matches_a\ f\ rs)$
 ⟨proof⟩

lemma *optimize-matches-a-simple-ruleset-eq*:
 $simple_ruleset\ rs \implies (\bigwedge m\ a. a = Accept \vee a = Drop \implies f1\ a\ m = f2\ a\ m) \implies$
 $optimize_matches_a\ f1\ rs = optimize_matches_a\ f2\ rs$
 ⟨proof⟩

lemma *optimize-matches-a-preserves*: $(\bigwedge r. r \in set\ rs \implies P\ (f\ (get_action\ r)\ (get_match\ r)))$
 $\implies \forall r \in set\ (optimize_matches_a\ f\ rs). P\ (get_match\ r)$
 ⟨proof⟩

end

theory *Semantics*

imports *Main Firewall-Common Common/List-Misc HOL-Library.LaTeXsugar*

begin

4 Big Step Semantics

The assumption we apply in general is that the firewall does not alter any packets.

A firewall ruleset is a map of chain names (e.g., INPUT, OUTPUT, FORWARD, arbitrary-user-defined-chain) to a list of rules. The list of rules is processed sequentially.

type-synonym $'a\ ruleset = string \rightarrow 'a\ rule\ list$

A matcher (parameterized by the type of primitive $'a$ and packet $'p$) is a function which just tells whether a given primitive and packet matches.

type-synonym $('a, 'p)\ matcher = 'a \Rightarrow 'p \Rightarrow bool$

Example: Assume a network packet only has a destination street number (for simplicity, of type nat) and we only support the following match expression: Is the packet's street number within a certain range? The type for the primitive could then be $nat \times nat$ and a possible implementation for $(nat \times nat, nat)\ matcher$ could be $match_street_number\ (a, b)\ p = (p \in \{a..b\})$. Usually, the primitives are a datatype which supports interfaces, IP addresses, protocols, ports, payload, ...

Given an $('a, 'p)\ matcher$ and a match expression, does a packet of type $'p$ match the match expression?

fun *matches* :: ('a, 'p) *matcher* ⇒ 'a *match-expr* ⇒ 'p ⇒ bool **where**
matches γ (*MatchAnd* e1 e2) p \longleftrightarrow *matches* γ e1 p \wedge *matches* γ e2 p |
matches γ (*MatchNot* me) p \longleftrightarrow \neg *matches* γ me p |
matches γ (*Match* e) p \longleftrightarrow γ e p |
matches - *MatchAny* - \longleftrightarrow True

inductive *iptables-bigstep* :: 'a *ruleset* ⇒ ('a, 'p) *matcher* ⇒ 'p ⇒ 'a *rule list* ⇒
state ⇒ *state* ⇒ bool

($\langle -, - \rangle$, $\langle -, - \rangle$) ⇒ \rightarrow [60,60,60,20,98,98] 89)

for Γ **and** γ **and** p **where**

skip: $\Gamma, \gamma, p \vdash \langle [], t \rangle \Rightarrow t$ |

accept: *matches* γ m p \Longrightarrow $\Gamma, \gamma, p \vdash \langle [Rule\ m\ Accept], Undecided \rangle \Rightarrow Decision\ FinalAllow$ |

drop: *matches* γ m p \Longrightarrow $\Gamma, \gamma, p \vdash \langle [Rule\ m\ Drop], Undecided \rangle \Rightarrow Decision\ FinalDeny$ |

reject: *matches* γ m p \Longrightarrow $\Gamma, \gamma, p \vdash \langle [Rule\ m\ Reject], Undecided \rangle \Rightarrow Decision\ FinalDeny$ |

log: *matches* γ m p \Longrightarrow $\Gamma, \gamma, p \vdash \langle [Rule\ m\ Log], Undecided \rangle \Rightarrow Undecided$ |

empty: *matches* γ m p \Longrightarrow $\Gamma, \gamma, p \vdash \langle [Rule\ m\ Empty], Undecided \rangle \Rightarrow Undecided$ |

nomatch: \neg *matches* γ m p \Longrightarrow $\Gamma, \gamma, p \vdash \langle [Rule\ m\ a], Undecided \rangle \Rightarrow Undecided$ |

decision: $\Gamma, \gamma, p \vdash \langle rs, Decision\ X \rangle \Rightarrow Decision\ X$ |

seq: $\llbracket \Gamma, \gamma, p \vdash \langle rs_1, Undecided \rangle \Rightarrow t; \Gamma, \gamma, p \vdash \langle rs_2, t \rangle \Rightarrow t' \rrbracket \Longrightarrow \Gamma, \gamma, p \vdash \langle rs_1 @ rs_2, Undecided \rangle \Rightarrow t'$ |

call-return: $\llbracket matches\ \gamma\ m\ p; \Gamma\ chain = Some\ (rs_1 @ [Rule\ m'\ Return] @ rs_2); matches\ \gamma\ m'\ p; \Gamma, \gamma, p \vdash \langle rs_1, Undecided \rangle \Rightarrow Undecided \rrbracket \Longrightarrow \Gamma, \gamma, p \vdash \langle [Rule\ m\ (Call\ chain)], Undecided \rangle \Rightarrow Undecided$ |

call-result: $\llbracket matches\ \gamma\ m\ p; \Gamma\ chain = Some\ rs; \Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow t \rrbracket \Longrightarrow$

$\Gamma, \gamma, p \vdash \langle [Rule\ m\ (Call\ chain)], Undecided \rangle \Rightarrow t$

The semantic rules again in pretty format:

$$\frac{\frac{\frac{\Gamma, \gamma, p \vdash \langle [], t \rangle \Rightarrow t}{matches\ \gamma\ m\ p}}{\Gamma, \gamma, p \vdash \langle [Rule\ m\ Accept], Undecided \rangle \Rightarrow Decision\ FinalAllow}}{matches\ \gamma\ m\ p}}{\Gamma, \gamma, p \vdash \langle [Rule\ m\ Drop], Undecided \rangle \Rightarrow Decision\ FinalDeny}}{matches\ \gamma\ m\ p}}{\Gamma, \gamma, p \vdash \langle [Rule\ m\ Reject], Undecided \rangle \Rightarrow Decision\ FinalDeny}}$$

$$\begin{array}{c}
\frac{\text{matches } \gamma \ m \ p}{\Gamma, \gamma, p \vdash \langle [Rule \ m \ Log], \ Undecided \rangle \Rightarrow \ Undecided} \\
\frac{\text{matches } \gamma \ m \ p}{\Gamma, \gamma, p \vdash \langle [Rule \ m \ Empty], \ Undecided \rangle \Rightarrow \ Undecided} \\
\frac{\neg \text{ matches } \gamma \ m \ p}{\Gamma, \gamma, p \vdash \langle [Rule \ m \ a], \ Undecided \rangle \Rightarrow \ Undecided} \\
\Gamma, \gamma, p \vdash \langle rs, \ Decision \ X \rangle \Rightarrow \ Decision \ X \\
\frac{\Gamma, \gamma, p \vdash \langle rs_1, \ Undecided \rangle \Rightarrow t \quad \Gamma, \gamma, p \vdash \langle rs_2, t \rangle \Rightarrow t'}{\Gamma, \gamma, p \vdash \langle rs_1 \ @ \ rs_2, \ Undecided \rangle \Rightarrow t'} \\
\frac{\text{matches } \gamma \ m \ p \quad \Gamma \ chain = \ Some \ (rs_1 \ @ \ [Rule \ m' \ Return] \ @ \ rs_2) \quad \text{matches } \gamma \ m' \ p \quad \Gamma, \gamma, p \vdash \langle rs_1, \ Undecided \rangle \Rightarrow \ Undecided}{\Gamma, \gamma, p \vdash \langle [Rule \ m \ (Call \ chain)], \ Undecided \rangle \Rightarrow \ Undecided} \\
\frac{\text{matches } \gamma \ m \ p \quad \Gamma \ chain = \ Some \ rs \quad \Gamma, \gamma, p \vdash \langle rs, \ Undecided \rangle \Rightarrow t}{\Gamma, \gamma, p \vdash \langle [Rule \ m \ (Call \ chain)], \ Undecided \rangle \Rightarrow t}
\end{array}$$

lemma deny:

$\text{matches } \gamma \ m \ p \implies a = \text{Drop} \vee a = \text{Reject} \implies \text{iptables-bigstep } \Gamma \ \gamma \ p \ [Rule \ m \ a] \ Undecided \ (Decision \ FinalDeny)$
⟨proof⟩

lemma seq-cons:

assumes $\Gamma, \gamma, p \vdash \langle [r], \ Undecided \rangle \Rightarrow t$ **and** $\Gamma, \gamma, p \vdash \langle rs, t \rangle \Rightarrow t'$
shows $\Gamma, \gamma, p \vdash \langle r \# rs, \ Undecided \rangle \Rightarrow t'$
⟨proof⟩

lemma iptables-bigstep-induct

[case-names Skip Allow Deny Log Nomatch Decision Seq Call-return Call-result,
induct pred: iptables-bigstep]:

$\llbracket \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t;$
 $\bigwedge t. P \llbracket t t;$
 $\bigwedge m \ a. \text{ matches } \gamma \ m \ p \implies a = \text{Accept} \implies P \ [Rule \ m \ a] \ Undecided \ (Decision \ FinalAllow);$
 $\bigwedge m \ a. \text{ matches } \gamma \ m \ p \implies a = \text{Drop} \vee a = \text{Reject} \implies P \ [Rule \ m \ a] \ Undecided \ (Decision \ FinalDeny);$
 $\bigwedge m \ a. \text{ matches } \gamma \ m \ p \implies a = \text{Log} \vee a = \text{Empty} \implies P \ [Rule \ m \ a] \ Undecided \ Undecided;$
 $\bigwedge m \ a. \neg \text{ matches } \gamma \ m \ p \implies P \ [Rule \ m \ a] \ Undecided \ Undecided;$
 $\bigwedge rs \ X. P \ rs \ (Decision \ X) \ (Decision \ X);$
 $\bigwedge rs \ rs_1 \ rs_2 \ t \ t'. rs = rs_1 \ @ \ rs_2 \implies \Gamma, \gamma, p \vdash \langle rs_1, \ Undecided \rangle \Rightarrow t \implies P \ rs_1 \ Undecided \ t \implies \Gamma, \gamma, p \vdash \langle rs_2, t \rangle \Rightarrow t' \implies P \ rs_2 \ t \ t' \implies P \ rs \ Undecided \ t';$
 $\bigwedge m \ a \ chain \ rs_1 \ m' \ rs_2. \text{ matches } \gamma \ m \ p \implies a = \text{Call chain} \implies \Gamma \ chain = \ Some \ (rs_1 \ @ \ [Rule \ m' \ Return] \ @ \ rs_2) \implies \text{ matches } \gamma \ m' \ p \implies \Gamma, \gamma, p \vdash \langle rs_1, \ Undecided \rangle \Rightarrow \ Undecided \implies P \ rs_1 \ Undecided \ Undecided \implies P \ [Rule \ m \ a] \ Undecided \ Undecided;$
 $\bigwedge m \ a \ chain \ rs \ t. \text{ matches } \gamma \ m \ p \implies a = \text{Call chain} \implies \Gamma \ chain = \ Some \ rs$

$\Longrightarrow \Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow t \Longrightarrow P \text{ rs Undecided } t \Longrightarrow P [\text{Rule } m \ a] \text{ Undecided } t \rceil \Longrightarrow$
 $P \text{ rs } s \ t$
 $\langle \text{proof} \rangle$

lemma skipD: $\Gamma, \gamma, p \vdash \langle r, s \rangle \Rightarrow t \Longrightarrow r = [] \Longrightarrow s = t$
 $\langle \text{proof} \rangle$

lemma decisionD: $\Gamma, \gamma, p \vdash \langle r, s \rangle \Rightarrow t \Longrightarrow s = \text{Decision } X \Longrightarrow t = \text{Decision } X$
 $\langle \text{proof} \rangle$

context

notes $\text{skipD}[\text{dest}] \ \text{list-app-singletonE}[\text{elim}]$

begin

lemma acceptD: $\Gamma, \gamma, p \vdash \langle r, s \rangle \Rightarrow t \Longrightarrow r = [\text{Rule } m \ \text{Accept}] \Longrightarrow \text{matches } \gamma \ m \ p \Longrightarrow$
 $s = \text{Undecided} \Longrightarrow t = \text{Decision } \text{FinalAllow}$
 $\langle \text{proof} \rangle$

lemma dropD: $\Gamma, \gamma, p \vdash \langle r, s \rangle \Rightarrow t \Longrightarrow r = [\text{Rule } m \ \text{Drop}] \Longrightarrow \text{matches } \gamma \ m \ p \Longrightarrow$
 $s = \text{Undecided} \Longrightarrow t = \text{Decision } \text{FinalDeny}$
 $\langle \text{proof} \rangle$

lemma rejectD: $\Gamma, \gamma, p \vdash \langle r, s \rangle \Rightarrow t \Longrightarrow r = [\text{Rule } m \ \text{Reject}] \Longrightarrow \text{matches } \gamma \ m \ p \Longrightarrow$
 $s = \text{Undecided} \Longrightarrow t = \text{Decision } \text{FinalDeny}$
 $\langle \text{proof} \rangle$

lemma logD: $\Gamma, \gamma, p \vdash \langle r, s \rangle \Rightarrow t \Longrightarrow r = [\text{Rule } m \ \text{Log}] \Longrightarrow \text{matches } \gamma \ m \ p \Longrightarrow s =$
 $\text{Undecided} \Longrightarrow t = \text{Undecided}$
 $\langle \text{proof} \rangle$

lemma emptyD: $\Gamma, \gamma, p \vdash \langle r, s \rangle \Rightarrow t \Longrightarrow r = [\text{Rule } m \ \text{Empty}] \Longrightarrow \text{matches } \gamma \ m \ p \Longrightarrow$
 $s = \text{Undecided} \Longrightarrow t = \text{Undecided}$
 $\langle \text{proof} \rangle$

lemma nomatchD: $\Gamma, \gamma, p \vdash \langle r, s \rangle \Rightarrow t \Longrightarrow r = [\text{Rule } m \ a] \Longrightarrow s = \text{Undecided} \Longrightarrow$
 $\neg \text{matches } \gamma \ m \ p \Longrightarrow t = \text{Undecided}$
 $\langle \text{proof} \rangle$

lemma callD:

assumes $\Gamma, \gamma, p \vdash \langle r, s \rangle \Rightarrow t \ r = [\text{Rule } m \ (\text{Call chain})] \ s = \text{Undecided} \ \text{matches } \gamma \ m \ p \ \Gamma \ \text{chain} = \text{Some } rs$

obtains $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$

$| \ rs_1 \ rs_2 \ m' \ \text{where } rs = rs_1 \ @ \ \text{Rule } m' \ \text{Return } \# \ rs_2 \ \text{matches } \gamma \ m' \ p \ \Gamma, \gamma, p \vdash$
 $\langle rs_1, s \rangle \Rightarrow \text{Undecided} \ t = \text{Undecided}$

$\langle \text{proof} \rangle$

end

lemmas *iptables-bigstepD = skipD acceptD dropD rejectD logD emptyD nomatchD decisionD callD*

lemma *seq'*:

assumes $rs = rs_1 @ rs_2 \quad \Gamma, \gamma, p \vdash \langle rs_1, s \rangle \Rightarrow t \quad \Gamma, \gamma, p \vdash \langle rs_2, t \rangle \Rightarrow t'$

shows $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t'$

<proof>

lemma *seq'-cons*: $\Gamma, \gamma, p \vdash \langle [r], s \rangle \Rightarrow t \Longrightarrow \Gamma, \gamma, p \vdash \langle rs, t \rangle \Rightarrow t' \Longrightarrow \Gamma, \gamma, p \vdash \langle r \# rs, s \rangle \Rightarrow t'$

<proof>

lemma *seq-split*:

assumes $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t \quad rs = rs_1 @ rs_2$

obtains t' **where** $\Gamma, \gamma, p \vdash \langle rs_1, s \rangle \Rightarrow t' \quad \Gamma, \gamma, p \vdash \langle rs_2, t' \rangle \Rightarrow t$

<proof>

lemma *seqE*:

assumes $\Gamma, \gamma, p \vdash \langle rs_1 @ rs_2, s \rangle \Rightarrow t$

obtains ti **where** $\Gamma, \gamma, p \vdash \langle rs_1, s \rangle \Rightarrow ti \quad \Gamma, \gamma, p \vdash \langle rs_2, ti \rangle \Rightarrow t$

<proof>

lemma *seqE-cons*:

assumes $\Gamma, \gamma, p \vdash \langle r \# rs, s \rangle \Rightarrow t$

obtains ti **where** $\Gamma, \gamma, p \vdash \langle [r], s \rangle \Rightarrow ti \quad \Gamma, \gamma, p \vdash \langle rs, ti \rangle \Rightarrow t$

<proof>

lemma *nomatch'*:

assumes $\bigwedge r. r \in \text{set } rs \Longrightarrow \neg \text{matches } \gamma (\text{get-match } r) p$

shows $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow s$

<proof>

there are only two cases when there can be a Return on top-level:

- the firewall is in a Decision state
- the return does not match

In both cases, it is not applied!

lemma *no-free-return*: **assumes** $\Gamma, \gamma, p \vdash \langle [\text{Rule } m \text{ Return}], \text{Undecided} \rangle \Rightarrow t$ **and** $\text{matches } \gamma m p$ **shows** *False*

<proof>

lemma *seq-progress*: $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t \Longrightarrow rs = rs_1 @ rs_2 \Longrightarrow \Gamma, \gamma, p \vdash \langle rs_1, s \rangle \Rightarrow t' \Longrightarrow \Gamma, \gamma, p \vdash \langle rs_2, t' \rangle \Rightarrow t$

<proof>

theorem *iptables-bigstep-deterministic*: **assumes** $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$ **and** $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t'$ **shows** $t = t'$
 ⟨proof⟩

lemma *iptables-bigstep-to-undecided*: $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow Undecided \Longrightarrow s = Undecided$
 ⟨proof⟩

lemma *iptables-bigstep-to-decision*: $\Gamma, \gamma, p \vdash \langle rs, Decision\ Y \rangle \Rightarrow Decision\ X \Longrightarrow Y = X$
 ⟨proof⟩

lemma *Rule-UndecidedE*:
assumes $\Gamma, \gamma, p \vdash \langle [Rule\ m\ a], Undecided \rangle \Rightarrow Undecided$
obtains $(nomatch) \neg matches\ \gamma\ m\ p$
 | $(log)\ a = Log \vee a = Empty$
 | $(call)\ c$ **where** $a = Call\ c\ matches\ \gamma\ m\ p$
 ⟨proof⟩

lemma *Rule-DecisionE*:
assumes $\Gamma, \gamma, p \vdash \langle [Rule\ m\ a], Undecided \rangle \Rightarrow Decision\ X$
obtains $(call)\ chain$ **where** $matches\ \gamma\ m\ p\ a = Call\ chain$
 | $(accept-reject)\ matches\ \gamma\ m\ p\ X = FinalAllow \Longrightarrow a = Accept\ X = FinalDeny \Longrightarrow a = Drop \vee a = Reject$
 ⟨proof⟩

lemma *log-remove*:
assumes $\Gamma, \gamma, p \vdash \langle rs_1 @ [Rule\ m\ Log] @ rs_2, s \rangle \Rightarrow t$
shows $\Gamma, \gamma, p \vdash \langle rs_1 @ rs_2, s \rangle \Rightarrow t$
 ⟨proof⟩

lemma *empty-empty*:
assumes $\Gamma, \gamma, p \vdash \langle rs_1 @ [Rule\ m\ Empty] @ rs_2, s \rangle \Rightarrow t$
shows $\Gamma, \gamma, p \vdash \langle rs_1 @ rs_2, s \rangle \Rightarrow t$
 ⟨proof⟩

lemma *Unknown-actions-False*: $\Gamma, \gamma, p \vdash \langle r \# rs, Undecided \rangle \Rightarrow t \Longrightarrow r = Rule\ m$
 $a \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow a = Unknown \vee (\exists chain. a = Goto\ chain) \Longrightarrow False$
 ⟨proof⟩

The notation we prefer in the paper. The semantics are defined for fixed Γ and γ

locale *iptables-bigstep-fixedbackground* =
fixes $\Gamma::'a\ ruleset$
and $\gamma::('a, 'p)\ matcher$
begin

inductive *iptables-bigstep'* :: 'a rule list \Rightarrow state \Rightarrow state \Rightarrow bool
 ($\langle +'' \langle -, - \rangle \Rightarrow \rightarrow$ [60,20,98,98] 89)
for *p* **where**
skip: $p \vdash' \langle [], t \rangle \Rightarrow t$ |
accept: $\text{matches } \gamma \ m \ p \Longrightarrow p \vdash' \langle [\text{Rule } m \ \text{Accept}], \text{Undecided} \rangle \Rightarrow \text{Decision FinalAllow}$ |
drop: $\text{matches } \gamma \ m \ p \Longrightarrow p \vdash' \langle [\text{Rule } m \ \text{Drop}], \text{Undecided} \rangle \Rightarrow \text{Decision FinalDeny}$
 |
reject: $\text{matches } \gamma \ m \ p \Longrightarrow p \vdash' \langle [\text{Rule } m \ \text{Reject}], \text{Undecided} \rangle \Rightarrow \text{Decision FinalDeny}$ |
log: $\text{matches } \gamma \ m \ p \Longrightarrow p \vdash' \langle [\text{Rule } m \ \text{Log}], \text{Undecided} \rangle \Rightarrow \text{Undecided}$ |
empty: $\text{matches } \gamma \ m \ p \Longrightarrow p \vdash' \langle [\text{Rule } m \ \text{Empty}], \text{Undecided} \rangle \Rightarrow \text{Undecided}$ |
nomatch: $\neg \text{matches } \gamma \ m \ p \Longrightarrow p \vdash' \langle [\text{Rule } m \ a], \text{Undecided} \rangle \Rightarrow \text{Undecided}$ |
decision: $p \vdash' \langle rs, \text{Decision } X \rangle \Rightarrow \text{Decision } X$ |
seq: $\llbracket p \vdash' \langle rs_1, \text{Undecided} \rangle \Rightarrow t; p \vdash' \langle rs_2, t \rangle \Rightarrow t' \rrbracket \Longrightarrow p \vdash' \langle rs_1 @ rs_2, \text{Undecided} \rangle \Rightarrow t'$ |
call-return: $\llbracket \text{matches } \gamma \ m \ p; \Gamma \ \text{chain} = \text{Some } (rs_1 @ [\text{Rule } m' \ \text{Return}] @ rs_2); \text{matches } \gamma \ m' \ p; p \vdash' \langle rs_1, \text{Undecided} \rangle \Rightarrow \text{Undecided} \rrbracket \Longrightarrow p \vdash' \langle [\text{Rule } m \ (\text{Call chain})], \text{Undecided} \rangle \Rightarrow \text{Undecided}$ |
call-result: $\llbracket \text{matches } \gamma \ m \ p; p \vdash' \langle \text{the } (\Gamma \ \text{chain}), \text{Undecided} \rangle \Rightarrow t \rrbracket \Longrightarrow p \vdash' \langle [\text{Rule } m \ (\text{Call chain})], \text{Undecided} \rangle \Rightarrow t$

definition *wf- Γ* :: 'a rule list \Rightarrow bool **where**
wf- Γ *rs* $\equiv \forall \text{rsg} \in \text{ran } \Gamma \cup \{rs\}. (\forall r \in \text{set } \text{rsg}. \forall \text{chain}. \text{get-action } r = \text{Call chain} \longrightarrow \Gamma \ \text{chain} \neq \text{None})$

lemma *wf- Γ -append*: $wf\text{-}\Gamma \ (rs1 @ rs2) \longleftrightarrow wf\text{-}\Gamma \ rs1 \wedge wf\text{-}\Gamma \ rs2$
 $\langle \text{proof} \rangle$

lemma *wf- Γ -tail*: $wf\text{-}\Gamma \ (r \# rs) \Longrightarrow wf\text{-}\Gamma \ rs$ $\langle \text{proof} \rangle$

lemma *wf- Γ -Call*: $wf\text{-}\Gamma \ [\text{Rule } m \ (\text{Call chain})] \Longrightarrow wf\text{-}\Gamma \ (\text{the } (\Gamma \ \text{chain})) \wedge (\exists rs. \Gamma \ \text{chain} = \text{Some } rs)$
 $\langle \text{proof} \rangle$

lemma *wf- Γ* $rs \Longrightarrow p \vdash' \langle rs, s \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, p \vdash' \langle rs, s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

end

Showing that semantics are defined. For rulesets which can be loaded by the Linux kernel. The kernel does not allow loops.

We call a ruleset well-formed (wf) iff all *Calls* are into actually existing chains.

definition *wf-chain* :: 'a ruleset \Rightarrow 'a rule list \Rightarrow bool **where**

wf-chain $\Gamma \ rs \equiv (\forall r \in \text{set } rs. \forall \text{chain}. \text{get-action } r = \text{Call chain} \longrightarrow \Gamma \ \text{chain} \neq \text{None})$

lemma *wf-chain-append*: $wf\text{-chain } \Gamma \ (rs1 @ rs2) \longleftrightarrow wf\text{-chain } \Gamma \ rs1 \wedge wf\text{-chain } \Gamma \ rs2$

<proof>

lemma *wf-chain-fst*: $wf-chain \Gamma (r \# rs) \implies wf-chain \Gamma (rs)$
<proof>

This is what our tool will check at runtime

definition *sanity-wf-ruleset* :: $(string \times 'a \text{ rule list}) list \Rightarrow bool$ **where**
 $sanity-wf-ruleset \Gamma \equiv distinct (map fst \Gamma) \wedge$
 $(\forall rs \in ran (map-of \Gamma). (\forall r \in set rs. case get-action r of Accept \Rightarrow True$
 $| Drop \Rightarrow True$
 $| Reject \Rightarrow True$
 $| Log \Rightarrow True$
 $| Empty \Rightarrow True$
 $| Call chain \Rightarrow chain \in dom$
 $| Goto chain \Rightarrow chain \in dom$
 $| Return \Rightarrow True$
 $| - \Rightarrow False))$

lemma *sanity-wf-ruleset-wf-chain*: $sanity-wf-ruleset \Gamma \implies rs \in ran (map-of \Gamma)$
 $\implies wf-chain (map-of \Gamma) rs$
<proof>

lemma *sanity-wf-ruleset-start*: $sanity-wf-ruleset \Gamma \implies chain-name \in dom (map-of \Gamma) \implies$
 $default-action = Accept \vee default-action = Drop \implies$
 $wf-chain (map-of \Gamma) [Rule MatchAny (Call chain-name), Rule MatchAny de-$
 $fault-action]$
<proof>

lemma [*code*]: $sanity-wf-ruleset \Gamma =$
 $(let dom = map fst \Gamma;$
 $ran = map snd \Gamma$
 $in distinct dom \wedge$
 $(\forall rs \in set ran. (\forall r \in set rs. case get-action r of Accept \Rightarrow True$
 $| Drop \Rightarrow True$
 $| Reject \Rightarrow True$
 $| Log \Rightarrow True$
 $| Empty \Rightarrow True$
 $| Call chain \Rightarrow chain \in set dom$
 $| Goto chain \Rightarrow chain \in set dom$
 $| Return \Rightarrow True$
 $| - \Rightarrow False)))$
<proof>

lemma semantics-bigstep-defined1: **assumes** $\forall rsg \in \text{ran } \Gamma \cup \{rs\}. \text{wf-chain } \Gamma \text{ rsg}$
and $\forall rsg \in \text{ran } \Gamma \cup \{rs\}. \forall r \in \text{set } rsg. (\forall \text{chain}. \text{get-action } r \neq \text{Goto chain}) \wedge$
 $\text{get-action } r \neq \text{Unknown}$
and $\forall r \in \text{set } rs. \text{get-action } r \neq \text{Return}$
and $(\forall \text{name} \in \text{dom } \Gamma. \exists t. \Gamma, \gamma, p \vdash \langle \text{the } (\Gamma \text{ name}), \text{Undecided} \rangle \Rightarrow t)$
shows $\exists t. \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

Showing the main theorem

context

begin

private lemma iptables-bigstep-defined-if-singleton-rules:

$\forall r \in \text{set } rs. (\exists t. \Gamma, \gamma, p \vdash \langle [r], s \rangle \Rightarrow t) \Longrightarrow \exists t. \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

well founded relation.

definition calls-chain :: 'a ruleset \Rightarrow (string \times string) set **where**

$\text{calls-chain } \Gamma = \{(r, s). \text{case } \Gamma \text{ r of Some } rs \Rightarrow \exists m. \text{Rule } m \text{ (Call } s) \in \text{set } rs \mid$
 $\text{None} \Rightarrow \text{False}\}$

lemma calls-chain-def2: $\text{calls-chain } \Gamma = \{(caller, callee). \exists rs \ m. \Gamma \text{ caller} =$
 $\text{Some } rs \wedge \text{Rule } m \text{ (Call } callee) \in \text{set } rs\}$

$\langle \text{proof} \rangle$

example

private lemma calls-chain [
 $\text{"FORWARD"} \mapsto [(\text{Rule } m1 \text{ Log}), (\text{Rule } m2 \text{ (Call "foo")}), (\text{Rule } m3 \text{ Accept}),$
 $(\text{Rule } m' \text{ (Call "baz")})],$
 $\text{"foo"} \mapsto [(\text{Rule } m4 \text{ Log}), (\text{Rule } m5 \text{ Return}), (\text{Rule } m6 \text{ (Call "bar")})],$
 $\text{"bar"} \mapsto [],$
 $\text{"baz"} \mapsto []] =$
 $\{(\text{"FORWARD"}, \text{"foo"}), (\text{"FORWARD"}, \text{"baz"}), (\text{"foo"}, \text{"bar"})\}$
 $\langle \text{proof} \rangle$ **lemma** *wf* ($\text{calls-chain } [$
 $\text{"FORWARD"} \mapsto [(\text{Rule } m1 \text{ Log}), (\text{Rule } m2 \text{ (Call "foo")}), (\text{Rule } m3 \text{ Accept}),$
 $(\text{Rule } m' \text{ (Call "baz")})],$
 $\text{"foo"} \mapsto [(\text{Rule } m4 \text{ Log}), (\text{Rule } m5 \text{ Return}), (\text{Rule } m6 \text{ (Call "bar")})],$
 $\text{"bar"} \mapsto [],$
 $\text{"baz"} \mapsto []])$
 $\langle \text{proof} \rangle$

In our proof, we will need the reverse.

private definition called-by-chain :: 'a ruleset \Rightarrow (string \times string) set **where**

$\text{called-by-chain } \Gamma = \{(callee, caller). \text{case } \Gamma \text{ caller of Some } rs \Rightarrow \exists m. \text{Rule } m$
 $(\text{Call } callee) \in \text{set } rs \mid \text{None} \Rightarrow \text{False}\}$

private lemma called-by-chain-converse: $\text{calls-chain } \Gamma = \text{converse } (\text{called-by-chain } \Gamma)$

$\langle \text{proof} \rangle$ **lemma** *wf-called-by-chain:* $\text{finite } (\text{calls-chain } \Gamma) \Longrightarrow \text{wf } (\text{calls-chain } \Gamma)$
 $\Longrightarrow \text{wf } (\text{called-by-chain } \Gamma)$

⟨proof⟩ **lemma** *helper-cases-call-subchain-defined-or-return*:
 $(\forall x \in \text{ran } \Gamma. \text{wf-chain } \Gamma x) \implies$
 $\forall \text{rsg} \in \text{ran } \Gamma. \forall r \in \text{set rsg}. (\forall \text{chain}. \text{get-action } r \neq \text{Goto chain}) \wedge \text{get-action}$
 $r \neq \text{Unknown} \implies$
 $\forall y m. \forall r \in \text{set rs-called}. r = \text{Rule } m (\text{Call } y) \longrightarrow (\exists t. \Gamma, \gamma, p \vdash \langle [\text{Rule } m$
 $(\text{Call } y)], \text{Undecided} \rangle \Rightarrow t) \implies$
 $\text{wf-chain } \Gamma \text{rs-called} \implies$
 $\forall r \in \text{set rs-called}. (\forall \text{chain}. \text{get-action } r \neq \text{Goto chain}) \wedge \text{get-action } r \neq$
 $\text{Unknown} \implies$
 $(\exists t. \Gamma, \gamma, p \vdash \langle \text{rs-called}, \text{Undecided} \rangle \Rightarrow t) \vee$
 $(\exists \text{rs-called1 rs-called2 } m'.$
 $\text{rs-called} = (\text{rs-called1} @ [\text{Rule } m' \text{Return}] @ \text{rs-called2}) \wedge$
 $\text{matches } \gamma m' p \wedge \Gamma, \gamma, p \vdash \langle \text{rs-called1}, \text{Undecided} \rangle \Rightarrow \text{Undecided})$
 ⟨proof⟩

lemma *helper-defined-single*:
assumes $\text{wf} (\text{called-by-chain } \Gamma)$
and $\forall \text{rsg} \in \text{ran } \Gamma \cup \{[\text{Rule } m a]\}. \text{wf-chain } \Gamma \text{rsg}$
and $\forall \text{rsg} \in \text{ran } \Gamma \cup \{[\text{Rule } m a]\}. \forall r \in \text{set rsg}. (\neg(\exists \text{chain}. \text{get-action } r =$
 $\text{Goto chain})) \wedge \text{get-action } r \neq \text{Unknown}$
and $a \neq \text{Return}$
shows $\exists t. \Gamma, \gamma, p \vdash \langle [\text{Rule } m a], s \rangle \Rightarrow t$
 ⟨proof⟩ **lemma** *helper-defined-ruleset-calledby*: $\text{wf} (\text{called-by-chain } \Gamma) \implies$
 $\forall \text{rsg} \in \text{ran } \Gamma \cup \{\text{rs}\}. \text{wf-chain } \Gamma \text{rsg} \implies$
 $\forall \text{rsg} \in \text{ran } \Gamma \cup \{\text{rs}\}. \forall r \in \text{set rsg}. (\neg(\exists \text{chain}. \text{get-action } r = \text{Goto chain})) \wedge$
 $\text{get-action } r \neq \text{Unknown} \implies$
 $\forall r \in \text{set rs}. \text{get-action } r \neq \text{Return} \implies$
 $\exists t. \Gamma, \gamma, p \vdash \langle \text{rs}, s \rangle \Rightarrow t$
 ⟨proof⟩

corollary *semantics-bigstep-defined*: $\text{finite} (\text{calls-chain } \Gamma) \implies \text{wf} (\text{calls-chain } \Gamma)$
 \implies — call relation finite and terminating
 $\forall \text{rsg} \in \text{ran } \Gamma \cup \{\text{rs}\}. \text{wf-chain } \Gamma \text{rsg} \implies$ — All calls to defined chains
 $\forall \text{rsg} \in \text{ran } \Gamma \cup \{\text{rs}\}. \forall r \in \text{set rsg}. (\forall x. \text{get-action } r \neq \text{Goto } x) \wedge \text{get-action}$
 $r \neq \text{Unknown} \implies$ — no bad actions
 $\forall r \in \text{set rs}. \text{get-action } r \neq \text{Return}$ — no toplevel return \implies
 $\exists t. \Gamma, \gamma, p \vdash \langle \text{rs}, s \rangle \Rightarrow t$
 ⟨proof⟩
end

Common Algorithms

lemma *iptables-bigstep-rm-LogEmpty*: $\Gamma, \gamma, p \vdash \langle \text{rm-LogEmpty } \text{rs}, s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash$
 $\langle \text{rs}, s \rangle \Rightarrow t$
 ⟨proof⟩

lemma *iptables-bigstep-rw-Reject*: $\Gamma, \gamma, p \vdash \langle \text{rw-Reject } \text{rs}, s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle \text{rs}, s \rangle$
 $\Rightarrow t$
 ⟨proof⟩

```

end
theory Matching
imports Semantics
begin

```

4.1 Boolean Matcher Algebra

lemma *MatchOr*: $\text{matches } \gamma \text{ (MatchOr } m1 \ m2) \ p \longleftrightarrow \text{matches } \gamma \ m1 \ p \vee \text{matches } \gamma \ m2 \ p$
 ⟨proof⟩

lemma *opt-MatchAny-match-expr-correct*: $\text{matches } \gamma \text{ (opt-MatchAny-match-expr } m) = \text{matches } \gamma \ m$
 ⟨proof⟩

lemma *matcheq-matchAny*: $\neg \text{has-primitive } m \implies \text{matcheq-matchAny } m \longleftrightarrow \text{matches } \gamma \ m \ p$
 ⟨proof⟩

lemma *matcheq-matchNone*: $\neg \text{has-primitive } m \implies \text{matcheq-matchNone } m \longleftrightarrow \neg \text{matches } \gamma \ m \ p$
 ⟨proof⟩

lemma *matcheq-matchNone-not-matches*: $\text{matcheq-matchNone } m \implies \neg \text{matches } \gamma \ m \ p$
 ⟨proof⟩

Lemmas about matching in the *iptables-bigstep* semantics.

lemma *matches-rule-iptables-bigstep*:
 assumes $\text{matches } \gamma \ m \ p \longleftrightarrow \text{matches } \gamma \ m' \ p$
 shows $\Gamma, \gamma, p \vdash \langle [\text{Rule } m \ a], s \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, p \vdash \langle [\text{Rule } m' \ a], s \rangle \Rightarrow t$ (is ?l \longleftrightarrow ?r)
 ⟨proof⟩

lemma *matches-rule-and-simp-help*:
 assumes $\text{matches } \gamma \ m \ p$
 shows $\Gamma, \gamma, p \vdash \langle [\text{Rule } (\text{MatchAnd } m \ m') \ a], \text{Undecided} \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, p \vdash \langle [\text{Rule } m' \ a], \text{Undecided} \rangle \Rightarrow t$ (is ?l \longleftrightarrow ?r)
 ⟨proof⟩

lemma *matches-MatchNot-simp*:
 assumes $\text{matches } \gamma \ m \ p$
 shows $\Gamma, \gamma, p \vdash \langle [\text{Rule } (\text{MatchNot } m) \ a], \text{Undecided} \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, p \vdash \langle [], \text{Undecided} \rangle \Rightarrow t$ (is ?l \longleftrightarrow ?r)
 ⟨proof⟩

lemma *matches-MatchNotAnd-simp*:

assumes *matches* γ m p

shows $\Gamma, \gamma, p \vdash \langle [Rule (MatchAnd (MatchNot m) m') a], Undecided \rangle \Rightarrow t \longleftrightarrow$
 $\Gamma, \gamma, p \vdash \langle [], Undecided \rangle \Rightarrow t$ (**is** $?l \longleftrightarrow ?r$)
<proof>

lemma *matches-rule-and-simp*:

assumes *matches* γ m p

shows $\Gamma, \gamma, p \vdash \langle [Rule (MatchAnd m m') a], s \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, p \vdash \langle [Rule m' a], s \rangle$
 $\Rightarrow t$
<proof>

lemma *iptables-bigstep-MatchAnd-comm*:

$\Gamma, \gamma, p \vdash \langle [Rule (MatchAnd m1 m2) a], s \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, p \vdash \langle [Rule (MatchAnd m2$
 $m1) a], s \rangle \Rightarrow t$
<proof>

4.2 Add match

definition *add-match* :: 'a match-expr \Rightarrow 'a rule list \Rightarrow 'a rule list **where**

add-match m $rs = \text{map } (\lambda r. \text{case } r \text{ of } Rule\ m'\ a' \Rightarrow Rule\ (MatchAnd\ m\ m')\ a')$
 rs

lemma *add-match-split*: *add-match* m ($rs1 @ rs2$) = *add-match* m $rs1$ @ *add-match*
 m $rs2$

<proof>

lemma *add-match-split-fst*: *add-match* m ($Rule\ m'\ a' \# rs$) = *Rule* (*MatchAnd* m
 m') $a' \#$ *add-match* m rs

<proof>

lemma *add-match-distrib*:

$\Gamma, \gamma, p \vdash \langle \text{add-match } m1\ (\text{add-match } m2\ rs), s \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, p \vdash \langle \text{add-match } m2$
 $(\text{add-match } m1\ rs), s \rangle \Rightarrow t$
<proof>

lemma *add-match-split-fst'*: *add-match* m ($a \# rs$) = *add-match* m $[a]$ @ *add-match*
 m rs

<proof>

lemma *matches-add-match-simp*:

assumes m : *matches* γ m p

shows $\Gamma, \gamma, p \vdash \langle \text{add-match } m\ rs, s \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$ (**is** $?l \longleftrightarrow ?r$)

<proof>

lemma *matches-add-match-MatchNot-simp*:

assumes m : *matches* γ m p
shows $\Gamma, \gamma, p \vdash \langle \text{add-match } (\text{MatchNot } m) \text{ } rs, s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle [], s \rangle \Rightarrow t$ (**is**
 $?l \ s \iff ?r \ s$)
 $\langle \text{proof} \rangle$

lemma *not-matches-add-match-simp*:

assumes $\neg \text{matches } \gamma \ m \ p$
shows $\Gamma, \gamma, p \vdash \langle \text{add-match } m \ rs, \text{Undecided} \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle [], \text{Undecided} \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

lemma *iptables-bigstep-add-match-notnot-simp*:

$\Gamma, \gamma, p \vdash \langle \text{add-match } (\text{MatchNot } (\text{MatchNot } m)) \ rs, s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle \text{add-match } m \ rs, s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

lemma *add-match-match-not-cases*:

$\Gamma, \gamma, p \vdash \langle \text{add-match } (\text{MatchNot } m) \ rs, \text{Undecided} \rangle \Rightarrow \text{Undecided} \implies \text{matches } \gamma \ m \ p \vee \Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Undecided}$
 $\langle \text{proof} \rangle$

lemma *not-matches-add-matchNot-simp*:

$\neg \text{matches } \gamma \ m \ p \implies \Gamma, \gamma, p \vdash \langle \text{add-match } (\text{MatchNot } m) \ rs, s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

lemma *iptables-bigstep-add-match-and*:

$\Gamma, \gamma, p \vdash \langle \text{add-match } m1 \ (\text{add-match } m2 \ rs), s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle \text{add-match } (\text{MatchAnd } m1 \ m2) \ rs, s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

lemma *optimize-matches-option-generic*:

assumes $\forall r \in \text{set } rs. P \ (\text{get-match } r)$
and $(\bigwedge m \ m'. P \ m \implies f \ m = \text{Some } m' \implies \text{matches } \gamma \ m' \ p = \text{matches } \gamma \ m \ p)$
and $(\bigwedge m. P \ m \implies f \ m = \text{None} \implies \neg \text{matches } \gamma \ m \ p)$
shows $\Gamma, \gamma, p \vdash \langle \text{optimize-matches-option } f \ rs, s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$
(is $?lhs \iff ?rhs$)
 $\langle \text{proof} \rangle$

lemma *optimize-matches-generic*: $\forall r \in \text{set } rs. P \ (\text{get-match } r) \implies$

$(\bigwedge m. P \ m \implies \text{matches } \gamma \ (f \ m) \ p = \text{matches } \gamma \ m \ p) \implies$

$\Gamma, \gamma, p \vdash \langle \text{optimize-matches } f \ rs, s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$

$\langle \text{proof} \rangle$

end

theory *Ruleset-Update*

imports *Matching*

begin

lemma *free-return-not-match*: $\Gamma, \gamma, p \vdash \langle [Rule\ m\ Return], Undecided \rangle \Rightarrow t \Longrightarrow \neg$
matches $\gamma\ m\ p$
<proof>

4.3 Background Ruleset Updating

lemma *update-Gamma-nomatch*:

assumes \neg *matches* $\gamma\ m\ p$
shows $\Gamma(chain \mapsto Rule\ m\ a \# rs), \gamma, p \vdash \langle rs', s \rangle \Rightarrow t \longleftrightarrow \Gamma(chain \mapsto rs), \gamma, p \vdash$
 $\langle rs', s \rangle \Rightarrow t$ (**is** $?l \longleftrightarrow ?r$)
<proof>

lemma *update-Gamma-log-empty*:

assumes $a = Log \vee a = Empty$
shows $\Gamma(chain \mapsto Rule\ m\ a \# rs), \gamma, p \vdash \langle rs', s \rangle \Rightarrow t \longleftrightarrow$
 $\Gamma(chain \mapsto rs), \gamma, p \vdash \langle rs', s \rangle \Rightarrow t$ (**is** $?l \longleftrightarrow ?r$)
<proof>

lemma *map-update-chain-if*: $(\lambda b. \text{if } b = chain \text{ then } Some\ rs \text{ else } \Gamma\ b) = \Gamma(chain$
 $\mapsto rs)$
<proof>

lemma *no-recursive-calls-helper*:

assumes $\Gamma, \gamma, p \vdash \langle [Rule\ m\ (Call\ chain)], Undecided \rangle \Rightarrow t$
and *matches* $\gamma\ m\ p$
and $\Gamma\ chain = Some\ [Rule\ m\ (Call\ chain)]$
shows *False*
<proof>

lemma *no-recursive-calls*:

$\Gamma(chain \mapsto [Rule\ m\ (Call\ chain)]), \gamma, p \vdash \langle [Rule\ m\ (Call\ chain)], Undecided \rangle \Rightarrow t$
 \Longrightarrow *matches* $\gamma\ m\ p \Longrightarrow$ *False*
<proof>

lemma *no-recursive-calls2*:

assumes $\Gamma(chain \mapsto (Rule\ m\ (Call\ chain)) \# rs''), \gamma, p \vdash \langle (Rule\ m\ (Call\ chain))$
 $\# rs', Undecided \rangle \Rightarrow Undecided$
and *matches* $\gamma\ m\ p$
shows *False*
<proof>

lemma *update-Gamma-nochange1*:

assumes $\Gamma(chain \mapsto rs), \gamma, p \vdash \langle [Rule\ m\ a], Undecided \rangle \Rightarrow Undecided$
and $\Gamma(chain \mapsto Rule\ m\ a \# rs), \gamma, p \vdash \langle rs', s \rangle \Rightarrow t$
shows $\Gamma(chain \mapsto rs), \gamma, p \vdash \langle rs', s \rangle \Rightarrow t$
<proof>

lemma *update-gamme-remove-Undecidedpart:*

assumes $\Gamma(\text{chain} \mapsto rs'), \gamma, p \vdash \langle rs', \text{Undecided} \rangle \Rightarrow \text{Undecided}$
and $\Gamma(\text{chain} \mapsto rs1@rs'), \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Undecided}$
shows $\Gamma(\text{chain} \mapsto rs'), \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Undecided}$
<proof>

lemma *update-Gamma-nocall:*

assumes $\neg (\exists \text{chain}. a = \text{Call chain})$
shows $\Gamma, \gamma, p \vdash \langle [\text{Rule } m \ a], s \rangle \Rightarrow t \longleftrightarrow \Gamma', \gamma, p \vdash \langle [\text{Rule } m \ a], s \rangle \Rightarrow t$
<proof>

lemma *update-Gamma-call:*

assumes $\Gamma \text{ chain} = \text{Some } rs$ **and** $\Gamma' \text{ chain} = \text{Some } rs'$
assumes $\Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Undecided}$ **and** $\Gamma', \gamma, p \vdash \langle rs', \text{Undecided} \rangle \Rightarrow \text{Undecided}$
shows $\Gamma, \gamma, p \vdash \langle [\text{Rule } m \ (\text{Call chain})], s \rangle \Rightarrow t \longleftrightarrow \Gamma', \gamma, p \vdash \langle [\text{Rule } m \ (\text{Call chain})], s \rangle \Rightarrow t$
<proof>

lemma *update-Gamma-remove-call-undecided:*

assumes $\Gamma(\text{chain} \mapsto \text{Rule } m \ (\text{Call } foo) \ \# \ rs'), \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Undecided}$
and *matches* $\gamma \ m \ p$
shows $\Gamma(\text{chain} \mapsto rs'), \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Undecided}$
<proof>

lemma *all-return-subchain:*

assumes $a1: \Gamma \text{ chain} = \text{Some } rs$
and $a2: \text{matches } \gamma \ m \ p$
and $a3: \forall r \in \text{set } rs. \text{get-action } r = \text{Return}$
shows $\Gamma, \gamma, p \vdash \langle [\text{Rule } m \ (\text{Call chain})], \text{Undecided} \rangle \Rightarrow \text{Undecided}$
<proof>

lemma *get-action-case-simp:* *get-action (case r of Rule m' x \Rightarrow Rule (MatchAnd m m') x) = get-action r*
<proof>

lemma *updategamma-insert-new:* $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t \implies \text{chain} \notin \text{dom } \Gamma \implies \Gamma(\text{chain} \mapsto rs'), \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$
<proof>

end

theory *Call-Return-Unfolding*

imports *Matching Ruleset-Update*

Common/Repeat-Stabilize
begin

5 Call Return Unfolding

Remove *Returns*

fun *process-ret* :: 'a rule list \Rightarrow 'a rule list **where**
process-ret [] = [] |
process-ret (Rule m Return # rs) = add-match (MatchNot m) (process-ret rs) |
process-ret (r#rs) = r # process-ret rs

Remove *Calls*

fun *process-call* :: 'a ruleset \Rightarrow 'a rule list \Rightarrow 'a rule list **where**
process-call Γ [] = [] |
process-call Γ (Rule m (Call chain) # rs) = add-match m (process-ret (the (Γ chain))) @ process-call Γ rs |
process-call Γ (r#rs) = r # process-call Γ rs

lemma *process-ret-split-fst-Return*:

$a = \text{Return} \implies \text{process-ret (Rule m a \# rs)} = \text{add-match (MatchNot m) (process-ret rs)}$
 <proof>

lemma *process-ret-split-fst-NeqReturn*:

$a \neq \text{Return} \implies \text{process-ret}((\text{Rule m a}) \# \text{rs}) = (\text{Rule m a}) \# (\text{process-ret rs})$
 <proof>

lemma *add-match-simp*: $\text{add-match m} = \text{map } (\lambda r. \text{Rule (MatchAnd m (get-match r)) (get-action r)})$
 <proof>

definition *add-missing-ret-unfoldings* :: 'a rule list \Rightarrow 'a rule list \Rightarrow 'a rule list
where

add-missing-ret-unfoldings rs1 rs2 \equiv
 foldr ($\lambda r f \text{ acc. add-match (MatchNot (get-match rf)) } \circ \text{acc}$) [$r \leftarrow \text{rs1. get-action } r = \text{Return}$] id rs2

fun *MatchAnd-foldr* :: 'a match-expr list \Rightarrow 'a match-expr **where**

MatchAnd-foldr [] = undefined |
MatchAnd-foldr [e] = e |
MatchAnd-foldr (e # es) = MatchAnd e (MatchAnd-foldr es)

fun *add-match-MatchAnd-foldr* :: 'a match-expr list \Rightarrow ('a rule list \Rightarrow 'a rule list)
where

add-match-MatchAnd-foldr [] = id |
add-match-MatchAnd-foldr es = add-match (MatchAnd-foldr es)

lemma *add-match-add-match-MatchAnd-foldr*:

$\Gamma, \gamma, p \vdash \langle \text{add-match } m \ (\text{add-match-MatchAnd-foldr } ms \ rs2), s \rangle \Rightarrow t = \Gamma, \gamma, p \vdash \langle \text{add-match } (\text{MatchAnd-foldr } (m \# ms)) \ rs2, s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

lemma *add-match-MatchAnd-foldr-empty-rs2*: $\text{add-match-MatchAnd-foldr } ms \ [] = []$
 $\langle \text{proof} \rangle$

lemma *add-missing-ret-unfoldings-alt*: $\Gamma, \gamma, p \vdash \langle \text{add-missing-ret-unfoldings } rs1 \ rs2, s \rangle \Rightarrow t \iff$
 $\Gamma, \gamma, p \vdash \langle (\text{add-match-MatchAnd-foldr } (\text{map } (\lambda r. \text{MatchNot } (\text{get-match } r))) [r \leftarrow rs1. \text{get-action } r = \text{Return}]]) \ rs2, s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

lemma *add-match-add-missing-ret-unfoldings-rot*:
 $\Gamma, \gamma, p \vdash \langle \text{add-match } m \ (\text{add-missing-ret-unfoldings } rs1 \ rs2), s \rangle \Rightarrow t =$
 $\Gamma, \gamma, p \vdash \langle \text{add-missing-ret-unfoldings } (\text{Rule } (\text{MatchNot } m) \ \text{Return} \# rs1) \ rs2, s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

5.1 Completeness

lemma *process-ret-split-obvious*: $\text{process-ret } (rs1 \ @ \ rs2) =$
 $(\text{process-ret } rs1) \ @ \ (\text{add-missing-ret-unfoldings } rs1 \ (\text{process-ret } rs2))$
 $\langle \text{proof} \rangle$

lemma *add-missing-ret-unfoldings-emptyrs2*: $\text{add-missing-ret-unfoldings } rs1 \ [] = []$
 $\langle \text{proof} \rangle$

lemma *process-call-split*: $\text{process-call } \Gamma \ (rs1 \ @ \ rs2) = \text{process-call } \Gamma \ rs1 \ @ \ \text{process-call } \Gamma \ rs2$
 $\langle \text{proof} \rangle$

lemma *process-call-split-fst*: $\text{process-call } \Gamma \ (a \ # \ rs) = \text{process-call } \Gamma \ [a] \ @ \ \text{process-call } \Gamma \ rs$
 $\langle \text{proof} \rangle$

lemma *iptables-bigstep-process-ret-undecided*: $\Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow t \implies \Gamma, \gamma, p \vdash \langle \text{process-ret } rs, \text{Undecided} \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

lemma *add-match-rot-add-missing-ret-unfoldings*:
 $\Gamma, \gamma, p \vdash \langle \text{add-match } m \ (\text{add-missing-ret-unfoldings } rs1 \ rs2), \text{Undecided} \rangle \Rightarrow \text{Undecided} =$
 $\Gamma, \gamma, p \vdash \langle \text{add-missing-ret-unfoldings } rs1 \ (\text{add-match } m \ rs2), \text{Undecided} \rangle \Rightarrow \text{Undecided}$

$\langle \text{proof} \rangle$

Completeness

theorem *unfolding-complete*: $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t \implies \Gamma, \gamma, p \vdash \langle \text{process-call } \Gamma \text{ } rs, s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

lemma *process-ret-cases*:

$\text{process-ret } rs = rs \vee (\exists rs_1 rs_2 m. rs = rs_1 @ [\text{Rule } m \text{ Return}] @ rs_2 \wedge (\text{process-ret } rs) = rs_1 @ (\text{process-ret } ([\text{Rule } m \text{ Return}] @ rs_2)))$
 $\langle \text{proof} \rangle$

lemma *process-ret-splitcases*:

obtains (id) $\text{process-ret } rs = rs$
| $(split)$ $rs_1 rs_2 m$ **where** $rs = rs_1 @ [\text{Rule } m \text{ Return}] @ rs_2$ **and** $\text{process-ret } rs = rs_1 @ (\text{process-ret } ([\text{Rule } m \text{ Return}] @ rs_2))$
 $\langle \text{proof} \rangle$

lemma *iptables-bigstep-process-ret-cases3*:

assumes $\Gamma, \gamma, p \vdash \langle \text{process-ret } rs, \text{Undecided} \rangle \Rightarrow \text{Undecided}$
obtains $(noreturn)$ $\Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Undecided}$
| $(return)$ $rs_1 rs_2 m$ **where** $rs = rs_1 @ [\text{Rule } m \text{ Return}] @ rs_2$ $\Gamma, \gamma, p \vdash \langle rs_1, \text{Undecided} \rangle \Rightarrow \text{Undecided}$ *matches* γ m p
 $\langle \text{proof} \rangle$

lemma *iptables-bigstep-process-ret-DecisionD*: $\Gamma, \gamma, p \vdash \langle \text{process-ret } rs, s \rangle \Rightarrow \text{Decision } X \implies \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow \text{Decision } X$
 $\langle \text{proof} \rangle$

5.2 process-ret correctness

lemma *process-ret-add-match-dist1*: $\Gamma, \gamma, p \vdash \langle \text{process-ret } (\text{add-match } m \text{ } rs), s \rangle \Rightarrow t \implies \Gamma, \gamma, p \vdash \langle \text{add-match } m \text{ } (\text{process-ret } rs), s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

lemma *process-ret-add-match-dist2*: $\Gamma, \gamma, p \vdash \langle \text{add-match } m \text{ } (\text{process-ret } rs), s \rangle \Rightarrow t \implies \Gamma, \gamma, p \vdash \langle \text{process-ret } (\text{add-match } m \text{ } rs), s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

lemma *process-ret-add-match-dist*: $\Gamma, \gamma, p \vdash \langle \text{process-ret } (\text{add-match } m \text{ } rs), s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle \text{add-match } m \text{ } (\text{process-ret } rs), s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

lemma *process-ret-Undecided-sound*:

assumes $\Gamma(\text{chain} \mapsto rs), \gamma, p \vdash \langle \text{process-ret } (\text{add-match } m \text{ } rs), \text{Undecided} \rangle \Rightarrow \text{Undecided}$

shows $\Gamma(\text{chain} \mapsto rs), \gamma, p \vdash \langle [\text{Rule } m \text{ } (\text{Call } \text{chain})], \text{Undecided} \rangle \Rightarrow \text{Undecided}$
 $\langle \text{proof} \rangle$

lemma *process-ret-Decision-sound*:

assumes $\Gamma(\text{chain} \mapsto rs), \gamma, p \vdash \langle \text{process-ret } (\text{add-match } m \text{ } rs), \text{Undecided} \rangle \Rightarrow \text{Decision } X$

shows $\Gamma(\text{chain} \mapsto rs), \gamma, p \vdash \langle [\text{Rule } m \text{ } (\text{Call } \text{chain})], \text{Undecided} \rangle \Rightarrow \text{Decision } X$
 $\langle \text{proof} \rangle$

lemma *process-ret-result-empty*: $\square = \text{process-ret } rs \Longrightarrow \forall r \in \text{set } rs. \text{get-action } r = \text{Return}$
 $\langle \text{proof} \rangle$

lemma *process-ret-sound'*:

assumes $\Gamma(\text{chain} \mapsto rs), \gamma, p \vdash \langle \text{process-ret } (\text{add-match } m \text{ } rs), \text{Undecided} \rangle \Rightarrow t$

shows $\Gamma(\text{chain} \mapsto rs), \gamma, p \vdash \langle [\text{Rule } m \text{ } (\text{Call } \text{chain})], \text{Undecided} \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

lemma *wf-chain-process-ret*: $\text{wf-chain } \Gamma \text{ } rs \Longrightarrow \text{wf-chain } \Gamma \text{ } (\text{process-ret } rs)$
 $\langle \text{proof} \rangle$

lemma *wf-chain-add-match*: $\text{wf-chain } \Gamma \text{ } rs \Longrightarrow \text{wf-chain } \Gamma \text{ } (\text{add-match } m \text{ } rs)$
 $\langle \text{proof} \rangle$

5.3 Soundness

theorem *unfolding-sound*: $\text{wf-chain } \Gamma \text{ } rs \Longrightarrow \Gamma, \gamma, p \vdash \langle \text{process-call } \Gamma \text{ } rs, s \rangle \Rightarrow t \Longrightarrow \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

corollary *unfolding-sound-complete*: $\text{wf-chain } \Gamma \text{ } rs \Longrightarrow \Gamma, \gamma, p \vdash \langle \text{process-call } \Gamma \text{ } rs, s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

corollary *unfolding-n-sound-complete*: $\forall rsg \in \text{ran } \Gamma \cup \{rs\}. \text{wf-chain } \Gamma \text{ } rsg \Longrightarrow \Gamma, \gamma, p \vdash \langle ((\text{process-call } \Gamma) \hat{\sim}^n) rs, s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

loops in the linux kernel:

```
http://lxr.linux.no/linux+v3.2/net/ipv4/netfilter/ip_tables.c#L464
/* Figures out from what hook each rule can be called: returns 0 if
   there are loops. Puts hook bitmask in comefrom. */
static int mark_source_chains(const struct xt_table_info *newinfo,
                              unsigned int valid_hooks, void *entry0)
```

discussion: <http://marc.info/?l=netfilter-devel&m=105190848425334&w=2>

Example

lemma *process-call* ["X" \mapsto [Rule (Match b) Return, Rule (Match c) Accept]] [Rule (Match a) (Call "X")] =
 [Rule (MatchAnd (Match a) (MatchAnd (MatchNot (Match b)) (Match c)))
 Accept] \langle proof \rangle

This is how a firewall processes a ruleset. It starts at a certain chain, usually INPUT, FORWARD, or OUTPUT (called *chain-name* in the lemma). The firewall has a default action of accept or drop. We can check *sanity-wf-ruleset* and the other assumptions at runtime. Consequently, we can apply *repeat-stabilize* as often as we want.

theorem *repeat-stabilize-process-call*:

assumes *sanity-wf-ruleset* Γ **and** *chain-name* \in set (map fst Γ) **and** *default-action* = Accept \vee *default-action* = Drop
shows (map-of Γ), γ , $p \vdash$ \langle repeat-stabilize n (process-call (map-of Γ)) [Rule MatchAny (Call *chain-name*), Rule MatchAny *default-action*], $s \rangle \Rightarrow t \iff$
 (map-of Γ), γ , $p \vdash$ \langle [Rule MatchAny (Call *chain-name*), Rule MatchAny *default-action*], $s \rangle \Rightarrow t$
 \langle proof \rangle

definition *unfold-optimize-ruleset-CHAIN*

$::$ ('a match-expr \Rightarrow 'a match-expr) \Rightarrow string \Rightarrow action \Rightarrow 'a ruleset \Rightarrow 'a rule list option

where

unfold-optimize-ruleset-CHAIN optimize *chain-name* *default-action* $rs =$ (let $rs =$
 (repeat-stabilize 1000 (optimize-matches opt-MatchAny-match-expr)
 (optimize-matches optimize
 (rw-Reject (rm-LogEmpty (repeat-stabilize 10000 (process-call rs)
 [Rule MatchAny (Call *chain-name*), Rule MatchAny *default-action*]
))))
 in if simple-ruleset rs then Some rs else None)

lemma *unfold-optimize-ruleset-CHAIN*:

assumes *sanity-wf-ruleset* Γ **and** *chain-name* \in set (map fst Γ)
and *default-action* = Accept \vee *default-action* = Drop
and $\bigwedge m$. matches γ (optimize m) $p =$ matches γ m p
and *unfold-optimize-ruleset-CHAIN* optimize *chain-name* *default-action* (map-of Γ) = Some rs
shows (map-of Γ), γ , $p \vdash$ \langle rs , $s \rangle \Rightarrow t \iff$
 (map-of Γ), γ , $p \vdash$ \langle [Rule MatchAny (Call *chain-name*), Rule MatchAny *default-action*], $s \rangle \Rightarrow t$
 \langle proof \rangle

end

6 Ternary Logic

```
theory Ternary
imports Main
begin
```

Kleene logic

```
datatype ternaryvalue = TernaryTrue | TernaryFalse | TernaryUnknown
datatype ternaryformula = TernaryAnd ternaryformula ternaryformula
                        | TernaryOr ternaryformula ternaryformula
                        | TernaryNot ternaryformula
                        | TernaryValue ternaryvalue
```

```
fun ternary-to-bool :: ternaryvalue  $\Rightarrow$  bool option where
  ternary-to-bool TernaryTrue = Some True |
  ternary-to-bool TernaryFalse = Some False |
  ternary-to-bool TernaryUnknown = None
```

```
fun bool-to-ternary :: bool  $\Rightarrow$  ternaryvalue where
  bool-to-ternary True = TernaryTrue |
  bool-to-ternary False = TernaryFalse
```

```
lemma the  $\circ$  ternary-to-bool  $\circ$  bool-to-ternary = id
  <proof>
```

```
lemma ternary-to-bool-bool-to-ternary: ternary-to-bool (bool-to-ternary X) = Some X
  <proof>
```

```
lemma ternary-to-bool-None: ternary-to-bool t = None  $\longleftrightarrow$  t = TernaryUnknown
  <proof>
```

```
lemma ternary-to-bool-SomeE: ternary-to-bool t = Some X  $\Longrightarrow$ 
  (t = TernaryTrue  $\Longrightarrow$  X = True  $\Longrightarrow$  P)  $\Longrightarrow$  (t = TernaryFalse  $\Longrightarrow$  X = False
 $\Longrightarrow$  P)  $\Longrightarrow$  P
  <proof>
```

```
lemma ternary-to-bool-Some: ternary-to-bool t = Some X  $\longleftrightarrow$ 
  (t = TernaryTrue  $\wedge$  X = True)  $\vee$  (t = TernaryFalse  $\wedge$  X = False)
  <proof>
```

```
lemma bool-to-ternary-Unknown: bool-to-ternary t = TernaryUnknown  $\longleftrightarrow$  False
  <proof>
```

```
fun eval-ternary-And :: ternaryvalue  $\Rightarrow$  ternaryvalue  $\Rightarrow$  ternaryvalue where
  eval-ternary-And TernaryTrue TernaryTrue = TernaryTrue |
  eval-ternary-And TernaryTrue TernaryFalse = TernaryFalse |
  eval-ternary-And TernaryFalse TernaryTrue = TernaryFalse |
  eval-ternary-And TernaryFalse TernaryFalse = TernaryFalse |
  eval-ternary-And TernaryFalse TernaryUnknown = TernaryFalse |
  eval-ternary-And TernaryTrue TernaryUnknown = TernaryUnknown |
  eval-ternary-And TernaryUnknown TernaryFalse = TernaryFalse |
```

eval-ternary-And TernaryUnknown TernaryTrue = TernaryUnknown |
eval-ternary-And TernaryUnknown TernaryUnknown = TernaryUnknown

lemma *eval-ternary-And-comm: eval-ternary-And t1 t2 = eval-ternary-And t2 t1*
 ⟨proof⟩

fun *eval-ternary-Or :: ternaryvalue ⇒ ternaryvalue ⇒ ternaryvalue where*
eval-ternary-Or TernaryTrue TernaryTrue = TernaryTrue |
eval-ternary-Or TernaryTrue TernaryFalse = TernaryTrue |
eval-ternary-Or TernaryFalse TernaryTrue = TernaryTrue |
eval-ternary-Or TernaryFalse TernaryFalse = TernaryFalse |
eval-ternary-Or TernaryTrue TernaryUnknown = TernaryTrue |
eval-ternary-Or TernaryFalse TernaryUnknown = TernaryUnknown |
eval-ternary-Or TernaryUnknown TernaryTrue = TernaryTrue |
eval-ternary-Or TernaryUnknown TernaryFalse = TernaryUnknown |
eval-ternary-Or TernaryUnknown TernaryUnknown = TernaryUnknown

fun *eval-ternary-Not :: ternaryvalue ⇒ ternaryvalue where*
eval-ternary-Not TernaryTrue = TernaryFalse |
eval-ternary-Not TernaryFalse = TernaryTrue |
eval-ternary-Not TernaryUnknown = TernaryUnknown

Just to hint that we did not make a typo, we add the truth table for the implication and show that it is compliant with $a \longrightarrow b = (\neg a \vee b)$

fun *eval-ternary-Imp :: ternaryvalue ⇒ ternaryvalue ⇒ ternaryvalue where*
eval-ternary-Imp TernaryTrue TernaryTrue = TernaryTrue |
eval-ternary-Imp TernaryTrue TernaryFalse = TernaryFalse |
eval-ternary-Imp TernaryFalse TernaryTrue = TernaryTrue |
eval-ternary-Imp TernaryFalse TernaryFalse = TernaryTrue |
eval-ternary-Imp TernaryTrue TernaryUnknown = TernaryUnknown |
eval-ternary-Imp TernaryFalse TernaryUnknown = TernaryTrue |
eval-ternary-Imp TernaryUnknown TernaryTrue = TernaryTrue |
eval-ternary-Imp TernaryUnknown TernaryFalse = TernaryUnknown |
eval-ternary-Imp TernaryUnknown TernaryUnknown = TernaryUnknown

lemma *eval-ternary-Imp a b = eval-ternary-Or (eval-ternary-Not a) b*
 ⟨proof⟩

lemma *eval-ternary-Not-UnknownD: eval-ternary-Not t = TernaryUnknown ⇒*
t = TernaryUnknown
 ⟨proof⟩

lemma *eval-ternary-DeMorgan:*
eval-ternary-Not (eval-ternary-And a b) = eval-ternary-Or (eval-ternary-Not a)
(eval-ternary-Not b)
eval-ternary-Not (eval-ternary-Or a b) = eval-ternary-And (eval-ternary-Not a)
(eval-ternary-Not b)
 ⟨proof⟩

lemma *eval-ternary-idempotence-Not*: *eval-ternary-Not (eval-ternary-Not a) = a*
 ⟨proof⟩

lemma *eval-ternary-simps-simple*:
eval-ternary-And TernaryTrue x = x
eval-ternary-And x TernaryTrue = x
eval-ternary-And TernaryFalse x = TernaryFalse
eval-ternary-And x TernaryFalse = TernaryFalse
 ⟨proof⟩

context

begin

private lemma *bool-to-ternary-simp1*: *bool-to-ternary X = TernaryTrue \longleftrightarrow X*
 ⟨proof⟩ **lemma** *bool-to-ternary-simp2*: *bool-to-ternary Y = TernaryFalse \longleftrightarrow*
 $\neg Y$
 ⟨proof⟩ **lemma** *bool-to-ternary-simp3*: *eval-ternary-Not (bool-to-ternary X) =*
TernaryTrue \longleftrightarrow $\neg X$
 ⟨proof⟩ **lemma** *bool-to-ternary-simp4*: *eval-ternary-Not (bool-to-ternary X) =*
TernaryFalse \longleftrightarrow X
 ⟨proof⟩ **lemma** *bool-to-ternary-simp5*: \neg (*eval-ternary-Not (bool-to-ternary X)*)
 $=$ *TernaryUnknown*
 ⟨proof⟩ **lemma** *bool-to-ternary-simp6*: *bool-to-ternary X \neq TernaryUnknown*
 ⟨proof⟩

lemmas *bool-to-ternary-simps = bool-to-ternary-simp1 bool-to-ternary-simp2*
bool-to-ternary-simp3 bool-to-ternary-simp4
bool-to-ternary-simp5 bool-to-ternary-simp6

end

context

begin

private lemma *bool-to-ternary-pullup1*:
eval-ternary-Not (bool-to-ternary X) = bool-to-ternary ($\neg X$)
 ⟨proof⟩ **lemma** *bool-to-ternary-pullup2*:
eval-ternary-And (bool-to-ternary X1) (bool-to-ternary X2) = bool-to-ternary
(X1 \wedge X2)
 ⟨proof⟩ **lemma** *bool-to-ternary-pullup3*:
eval-ternary-Imp (bool-to-ternary X1) (bool-to-ternary X2) = bool-to-ternary
(X1 \longrightarrow X2)
 ⟨proof⟩ **lemma** *bool-to-ternary-pullup4*:
eval-ternary-Or (bool-to-ternary X1) (bool-to-ternary X2) = bool-to-ternary (X1
 \vee *X2)*
 ⟨proof⟩

lemmas *bool-to-ternary-pullup = bool-to-ternary-pullup1 bool-to-ternary-pullup2*
bool-to-ternary-pullup3 bool-to-ternary-pullup4

end

fun *ternary-ternary-eval* :: *ternaryformula* \Rightarrow *ternaryvalue* **where**
 ternary-ternary-eval (*TernaryAnd* *t1* *t2*) = *eval-ternary-And* (*ternary-ternary-eval* *t1*) (*ternary-ternary-eval* *t2*) |
 ternary-ternary-eval (*TernaryOr* *t1* *t2*) = *eval-ternary-Or* (*ternary-ternary-eval* *t1*) (*ternary-ternary-eval* *t2*) |
 ternary-ternary-eval (*TernaryNot* *t*) = *eval-ternary-Not* (*ternary-ternary-eval* *t*)
 |
 ternary-ternary-eval (*TernaryValue* *t*) = *t*

lemma *ternary-ternary-eval-DeMorgan*: *ternary-ternary-eval* (*TernaryNot* (*TernaryAnd* *a* *b*)) =
 ternary-ternary-eval (*TernaryOr* (*TernaryNot* *a*) (*TernaryNot* *b*))
{*proof*}

lemma *ternary-ternary-eval-idempotence-Not*:
 ternary-ternary-eval (*TernaryNot* (*TernaryNot* *a*)) = *ternary-ternary-eval* *a*
{*proof*}

lemma *ternary-ternary-eval-TernaryAnd-comm*:
 ternary-ternary-eval (*TernaryAnd* *t1* *t2*) = *ternary-ternary-eval* (*TernaryAnd* *t2* *t1*)
{*proof*}

lemma *eval-ternary-Not* (*ternary-ternary-eval* *t*) = (*ternary-ternary-eval* (*TernaryNot* *t*)) {*proof*}

context

begin

private lemma *eval-ternary-simps-2*:

eval-ternary-And (*bool-to-ternary* *P*) *T* = *TernaryTrue* \longleftrightarrow *P* \wedge *T* = *Ternary-True*

eval-ternary-And *T* (*bool-to-ternary* *P*) = *TernaryTrue* \longleftrightarrow *P* \wedge *T* = *Ternary-True*

 {*proof*} **lemma** *eval-ternary-simps-3*:

eval-ternary-And (*ternary-ternary-eval* *x*) *T* = *TernaryTrue* \longleftrightarrow

ternary-ternary-eval *x* = *TernaryTrue* \wedge *T* = *TernaryTrue*

eval-ternary-And *T* (*ternary-ternary-eval* *x*) = *TernaryTrue* \longleftrightarrow

ternary-ternary-eval *x* = *TernaryTrue* \wedge *T* = *TernaryTrue*

 {*proof*}

lemmas *eval-ternary-simps* = *eval-ternary-simps-simple* *eval-ternary-simps-2* *eval-ternary-simps-3*

end

definition *ternary-eval* :: *ternaryformula* \Rightarrow *bool option* **where**
ternary-eval *t* = *ternary-to-bool* (*ternary-ternary-eval* *t*)

6.1 Negation Normal Form

A formula is in Negation Normal Form (NNF) if negations only occur at the atoms (not before and/or)

inductive *NegationNormalForm* :: *ternaryformula* \Rightarrow *bool* **where**
NegationNormalForm (*TernaryValue* *v*) |
NegationNormalForm (*TernaryNot* (*TernaryValue* *v*)) |
NegationNormalForm $\varphi \Longrightarrow$ *NegationNormalForm* $\psi \Longrightarrow$ *NegationNormalForm*
(*TernaryAnd* φ ψ) |
NegationNormalForm $\varphi \Longrightarrow$ *NegationNormalForm* $\psi \Longrightarrow$ *NegationNormalForm*
(*TernaryOr* φ ψ)

Convert a *ternaryformula* to a *ternaryformula* in NNF.

fun *NNF-ternary* :: *ternaryformula* \Rightarrow *ternaryformula* **where**
NNF-ternary (*TernaryValue* *v*) = *TernaryValue* *v* |
NNF-ternary (*TernaryAnd* *t1* *t2*) = *TernaryAnd* (*NNF-ternary* *t1*) (*NNF-ternary*
t2) |
NNF-ternary (*TernaryOr* *t1* *t2*) = *TernaryOr* (*NNF-ternary* *t1*) (*NNF-ternary*
t2) |
NNF-ternary (*TernaryNot* (*TernaryNot* *t*)) = *NNF-ternary* *t* |
NNF-ternary (*TernaryNot* (*TernaryValue* *v*)) = *TernaryValue* (*eval-ternary-Not*
v) |
NNF-ternary (*TernaryNot* (*TernaryAnd* *t1* *t2*)) = *TernaryOr* (*NNF-ternary*
(*TernaryNot* *t1*)) (*NNF-ternary* (*TernaryNot* *t2*)) |
NNF-ternary (*TernaryNot* (*TernaryOr* *t1* *t2*)) = *TernaryAnd* (*NNF-ternary*
(*TernaryNot* *t1*)) (*NNF-ternary* (*TernaryNot* *t2*))

lemma *NNF-ternary-correct*: *ternary-ternary-eval* (*NNF-ternary* *t*) = *ternary-ternary-eval*
t
<proof>

lemma *NNF-ternary-NegationNormalForm*: *NegationNormalForm* (*NNF-ternary*
t)
<proof>

context
begin

private lemma *ternary-lift1*: *eval-ternary-Not* *tv* \neq *TernaryFalse* \longleftrightarrow *tv* =
TernaryFalse \vee *tv* = *TernaryUnknown*
<proof> **lemma** *ternary-lift2*: *eval-ternary-Not* *tv* \neq *TernaryTrue* \longleftrightarrow *tv* =
TernaryTrue \vee *tv* = *TernaryUnknown*

```

    <proof> lemma ternary-lift3: eval-ternary-Not tv = TernaryFalse  $\longleftrightarrow$  tv =
TernaryTrue
    <proof> lemma ternary-lift4: eval-ternary-Not tv = TernaryTrue  $\longleftrightarrow$  tv =
TernaryFalse
    <proof> lemma ternary-lift5: eval-ternary-Not tv = TernaryUnknown  $\longleftrightarrow$  tv
= TernaryUnknown
    <proof> lemma ternary-lift6: eval-ternary-And t1 t2 = TernaryFalse  $\longleftrightarrow$  t1 =
TernaryFalse  $\vee$  t2 = TernaryFalse
    <proof> lemma ternary-lift7: eval-ternary-And t1 t2 = TernaryTrue  $\longleftrightarrow$  t1 =
TernaryTrue  $\wedge$  t2 = TernaryTrue
    <proof>

```

```

lemmas ternary-lift = ternary-lift1 ternary-lift2 ternary-lift3 ternary-lift4 ternary-lift5
ternary-lift6 ternary-lift7
end

```

```

context

```

```

begin

```

```

    private lemma l1: eval-ternary-Not tv = TernaryTrue  $\implies$  tv = TernaryFalse
    <proof> lemma l2: eval-ternary-And t1 t2 = TernaryFalse  $\implies$  t1 = Ternary-
False  $\vee$  t2 = TernaryFalse
    <proof>

```

```

lemmas eval-ternaryD = l1 l2
end

```

```

end

```

```

theory Matching-Ternary

```

```

imports ../Common/Ternary ../Firewall-Common

```

```

begin

```

7 Packet Matching in Ternary Logic

The matcher for a primitive match expression $'a$

```

type-synonym ('a, 'packet) exact-match-tac= $'a \Rightarrow 'packet \Rightarrow ternaryvalue$ 

```

If the matching is *TernaryUnknown*, it can be decided by the action whether this rule matches. E.g. in doubt, we allow packets

```

type-synonym 'packet unknown-match-tac= $action \Rightarrow 'packet \Rightarrow bool$ 

```

```

type-synonym ('a, 'packet) match-tac= $(('a, 'packet) exact-match-tac \times 'packet$ 
unknown-match-tac)

```

For a given packet, map a firewall $'a$ match-expr to a *ternaryformula* Evaluating the formula gives whether the packet/rule matches (or unknown).

```

fun map-match-tac :: ('a, 'packet) exact-match-tac  $\Rightarrow 'packet \Rightarrow 'a$  match-expr  $\Rightarrow$ 
ternaryformula where

```

```

map-match-tac β p (MatchAnd m1 m2) = TernaryAnd (map-match-tac β p m1)
(map-match-tac β p m2) |
map-match-tac β p (MatchNot m) = TernaryNot (map-match-tac β p m) |
map-match-tac β p (Match m) = TernaryValue (β m p) |
map-match-tac - - MatchAny = TernaryValue TernaryTrue

```

context
begin

the *ternaryformulas* we construct never have Or expressions.

```

private fun ternary-has-or :: ternaryformula ⇒ bool where
  ternary-has-or (TernaryOr - -) ←→ True |
  ternary-has-or (TernaryAnd t1 t2) ←→ ternary-has-or t1 ∨ ternary-has-or t2
|
  ternary-has-or (TernaryNot t) ←→ ternary-has-or t |
  ternary-has-or (TernaryValue -) ←→ False
private lemma map-match-tac--does-not-use-TernaryOr: ¬ (ternary-has-or (map-match-tac
β p m))
  ⟨proof⟩
declare ternary-has-or.simps[simp del]
end

```

```

fun ternary-to-bool-unknown-match-tac :: 'a packet unknown-match-tac ⇒ action ⇒
'packet ⇒ ternaryvalue ⇒ bool where
  ternary-to-bool-unknown-match-tac - - - TernaryTrue = True |
  ternary-to-bool-unknown-match-tac - - - TernaryFalse = False |
  ternary-to-bool-unknown-match-tac α a p TernaryUnknown = α a p

```

Matching a packet and a rule:

1. Translate '*a match-expr*' to ternary formula
2. Evaluate this formula
3. If *TernaryTrue/TernaryFalse*, return this value
4. If *TernaryUnknown*, apply the '*a unknown-match-tac*' to get a Boolean result

```

definition matches :: ('a, 'packet) match-tac ⇒ 'a match-expr ⇒ action ⇒ 'packet
⇒ bool where
  matches γ m a p ≡ ternary-to-bool-unknown-match-tac (snd γ) a p (ternary-ternary-eval
(map-match-tac (fst γ) p m))

```

Alternative matches definitions, some more or less convenient

```

lemma matches-tuple: matches (β, α) m a p = ternary-to-bool-unknown-match-tac
α a p (ternary-ternary-eval (map-match-tac β p m))

```

<proof>

lemma *matches-case*: $\text{matches } \gamma \ m \ a \ p \longleftrightarrow (\text{case ternary-eval } (\text{map-match-tac } (\text{fst } \gamma) \ p \ m) \ \text{of } \text{None} \Rightarrow (\text{snd } \gamma) \ a \ p \mid \text{Some } b \Rightarrow b)$
<proof>

lemma *matches-case-tuple*: $\text{matches } (\beta, \alpha) \ m \ a \ p \longleftrightarrow (\text{case ternary-eval } (\text{map-match-tac } \beta \ p \ m) \ \text{of } \text{None} \Rightarrow \alpha \ a \ p \mid \text{Some } b \Rightarrow b)$
<proof>

lemma *matches-case-ternaryvalue-tuple*: $\text{matches } (\beta, \alpha) \ m \ a \ p \longleftrightarrow (\text{case ternary-ternary-eval } (\text{map-match-tac } \beta \ p \ m) \ \text{of } \text{TernaryUnknown} \Rightarrow \alpha \ a \ p \mid \text{TernaryTrue} \Rightarrow \text{True} \mid \text{TernaryFalse} \Rightarrow \text{False})$
<proof>

lemma *matches-casesE*:
 $\text{matches } (\beta, \alpha) \ m \ a \ p \Longrightarrow (\text{ternary-ternary-eval } (\text{map-match-tac } \beta \ p \ m) = \text{TernaryUnknown} \Longrightarrow \alpha \ a \ p \Longrightarrow P) \Longrightarrow (\text{ternary-ternary-eval } (\text{map-match-tac } \beta \ p \ m) = \text{TernaryTrue} \Longrightarrow P) \Longrightarrow P$
<proof>

Example: $\neg \text{Unknown}$ is as good as Unknown

lemma $\llbracket \text{ternary-ternary-eval } (\text{map-match-tac } \beta \ p \ \text{expr}) = \text{TernaryUnknown} \rrbracket \Longrightarrow \text{matches } (\beta, \alpha) \ \text{expr} \ a \ p \longleftrightarrow \text{matches } (\beta, \alpha) \ (\text{MatchNot expr}) \ a \ p$
<proof>

lemma *bunch-of-lemmata-about-matches*:
 $\text{matches } \gamma \ (\text{MatchAnd } m1 \ m2) \ a \ p \longleftrightarrow \text{matches } \gamma \ m1 \ a \ p \wedge \text{matches } \gamma \ m2 \ a \ p$
 $\text{matches } \gamma \ \text{MatchAny} \ a \ p$
 $\text{matches } \gamma \ (\text{MatchNot MatchAny}) \ a \ p \longleftrightarrow \text{False}$
 $\text{matches } \gamma \ (\text{MatchNot } (\text{MatchNot } m)) \ a \ p \longleftrightarrow \text{matches } \gamma \ m \ a \ p$
<proof>

lemma *match-raw-bool*:
 $\text{matches } (\beta, \alpha) \ (\text{Match expr}) \ a \ p = (\text{case ternary-to-bool } (\beta \ \text{expr} \ p) \ \text{of } \text{Some } r \Rightarrow r \mid \text{None} \Rightarrow (\alpha \ a \ p))$
<proof>

lemma *match-raw-ternary*:
 $\text{matches } (\beta, \alpha) \ (\text{Match expr}) \ a \ p = (\text{case } (\beta \ \text{expr} \ p) \ \text{of } \text{TernaryTrue} \Rightarrow \text{True} \mid \text{TernaryFalse} \Rightarrow \text{False} \mid \text{TernaryUnknown} \Rightarrow (\alpha \ a \ p))$
<proof>

lemma *matches-DeMorgan*: $\text{matches } \gamma \text{ (MatchNot (MatchAnd m1 m2)) } a \text{ } p \longleftrightarrow$
 $(\text{matches } \gamma \text{ (MatchNot m1) } a \text{ } p) \vee (\text{matches } \gamma \text{ (MatchNot m2) } a \text{ } p)$
 $\langle \text{proof} \rangle$

7.1 Ternary Matcher Algebra

lemma *matches-and-comm*: $\text{matches } \gamma \text{ (MatchAnd m m') } a \text{ } p \longleftrightarrow \text{matches } \gamma$
 $\text{(MatchAnd m' m) } a \text{ } p$
 $\langle \text{proof} \rangle$

lemma *matches-not-idem*: $\text{matches } \gamma \text{ (MatchNot (MatchNot m)) } a \text{ } p \longleftrightarrow \text{matches}$
 $\gamma \text{ m } a \text{ } p$
 $\langle \text{proof} \rangle$

lemma *MatchOr*: $\text{matches } \gamma \text{ (MatchOr m1 m2) } a \text{ } p \longleftrightarrow \text{matches } \gamma \text{ m1 } a \text{ } p \vee$
 $\text{matches } \gamma \text{ m2 } a \text{ } p$
 $\langle \text{proof} \rangle$

lemma *MatchOr-MatchNot*: $\text{matches } \gamma \text{ (MatchNot (MatchOr m1 m2)) } a \text{ } p \longleftrightarrow$
 $\text{matches } \gamma \text{ (MatchNot m1) } a \text{ } p \wedge \text{matches } \gamma \text{ (MatchNot m2) } a \text{ } p$
 $\langle \text{proof} \rangle$

lemma *(TernaryNot (map-match-tac β p (m))) = (map-match-tac β p (MatchNot*
 $m))$
 $\langle \text{proof} \rangle$

context

begin

private lemma *matches-simp1*: $\text{matches } \gamma \text{ m } a \text{ } p \implies \text{matches } \gamma \text{ (MatchAnd m}$
 $m') \text{ } a \text{ } p \longleftrightarrow \text{matches } \gamma \text{ m' } a \text{ } p$

$\langle \text{proof} \rangle$ **lemma** *matches-simp11*: $\text{matches } \gamma \text{ m } a \text{ } p \implies \text{matches } \gamma \text{ (MatchAnd}$
 $m' m) \text{ } a \text{ } p \longleftrightarrow \text{matches } \gamma \text{ m' } a \text{ } p$

$\langle \text{proof} \rangle$ **lemma** *matches-simp2*: $\text{matches } \gamma \text{ (MatchAnd m m') } a \text{ } p \implies \neg \text{matches}$
 $\gamma \text{ m } a \text{ } p \implies \text{False}$

$\langle \text{proof} \rangle$ **lemma** *matches-simp22*: $\text{matches } \gamma \text{ (MatchAnd m m') } a \text{ } p \implies \neg$
 $\text{matches } \gamma \text{ m' } a \text{ } p \implies \text{False}$

$\langle \text{proof} \rangle$ **lemma** *matches-simp3*: $\text{matches } \gamma \text{ (MatchNot m) } a \text{ } p \implies \text{matches } \gamma$
 $m \text{ } a \text{ } p \implies (\text{snd } \gamma) \text{ } a \text{ } p$

$\langle \text{proof} \rangle$ **lemma** *matches* $\gamma \text{ (MatchNot m) } a \text{ } p \implies \text{matches } \gamma \text{ m } a \text{ } p \implies$
 $(\text{ternary-eval (map-match-tac (fst } \gamma) \text{ p m)) = None}$

$\langle \text{proof} \rangle$

lemmas *matches-simps* = *matches-simp1 matches-simp11*

lemmas *matches-dest* = *matches-simp2 matches-simp22*

end

lemma *matches-iff-apply-f-generic: ternary-ternary-eval (map-match-tac β p (f (β, α) a m)) = ternary-ternary-eval (map-match-tac β p m) \implies matches (β, α) (f (β, α) a m) a p \iff matches (β, α) m a p*
 ⟨proof⟩

lemma *matches-iff-apply-f: ternary-ternary-eval (map-match-tac β p (f m)) = ternary-ternary-eval (map-match-tac β p m) \implies matches (β, α) (f m) a p \iff matches (β, α) m a p*
 ⟨proof⟩

lemma *opt-MatchAny-match-expr-correct: matches γ (opt-MatchAny-match-expr m) = matches γ m*
 ⟨proof⟩

An *'p unknown-match-tac* is wf if it behaves equal for *Reject* and *Drop*

definition *wf-unknown-match-tac :: 'p unknown-match-tac \Rightarrow bool where*
wf-unknown-match-tac $\alpha \equiv (\alpha$ Drop = α Reject)

lemma *wf-unknown-match-tacD-False1: wf-unknown-match-tac $\alpha \implies \neg$ matches (β, α) m Reject $p \implies$ matches (β, α) m Drop $p \implies$ False*
 ⟨proof⟩

lemma *wf-unknown-match-tacD-False2: wf-unknown-match-tac $\alpha \implies$ matches (β, α) m Reject $p \implies \neg$ matches (β, α) m Drop $p \implies$ False*
 ⟨proof⟩

7.2 Removing Unknown Primitives

definition *unknown-match-all :: 'a unknown-match-tac \Rightarrow action \Rightarrow bool where*
unknown-match-all α $a = (\forall p. \alpha$ a $p)$

definition *unknown-not-match-any :: 'a unknown-match-tac \Rightarrow action \Rightarrow bool where*
unknown-not-match-any α $a = (\forall p. \neg \alpha$ a $p)$

fun *remove-unknowns-generic :: ('a, 'packet) match-tac \Rightarrow action \Rightarrow 'a match-expr \Rightarrow 'a match-expr where*

remove-unknowns-generic - - MatchAny = MatchAny |
remove-unknowns-generic - - (MatchNot MatchAny) = MatchNot MatchAny |
remove-unknowns-generic (β, α) a (Match A) = (if
($\forall p. \text{ternary-ternary-eval (map-match-tac } \beta$ p (Match A)) = TernaryUnknown)
then
if unknown-match-all α a then MatchAny else if unknown-not-match-any α a
then MatchNot MatchAny else Match A
else (Match A)) |

remove-unknowns-generic $(\beta, \alpha) a$ (*MatchNot* (*Match* A)) = (if
 $(\forall p. \text{ternary-ternary-eval} (\text{map-match-tac } \beta p (\text{Match } A)) = \text{TernaryUnknown})$
then
if *unknown-match-all* αa then *MatchAny* else if *unknown-not-match-any* αa
then *MatchNot MatchAny* else *MatchNot* (*Match* A)
else *MatchNot* (*Match* A)) |
remove-unknowns-generic $(\beta, \alpha) a$ (*MatchNot* (*MatchNot* m)) = *remove-unknowns-generic*
 $(\beta, \alpha) a m$ |
remove-unknowns-generic $(\beta, \alpha) a$ (*MatchAnd* $m1 m2$) = *MatchAnd*
(*remove-unknowns-generic* $(\beta, \alpha) a m1$)
(*remove-unknowns-generic* $(\beta, \alpha) a m2$) |
— $\neg (a \wedge b) = \neg b \vee \neg a$ and $\neg \text{Unknown} = \text{Unknown}$
remove-unknowns-generic $(\beta, \alpha) a$ (*MatchNot* (*MatchAnd* $m1 m2$)) =
(if (*remove-unknowns-generic* $(\beta, \alpha) a$ (*MatchNot* $m1$)) = *MatchAny* \vee
(*remove-unknowns-generic* $(\beta, \alpha) a$ (*MatchNot* $m2$)) = *MatchAny*
then *MatchAny* else
(if (*remove-unknowns-generic* $(\beta, \alpha) a$ (*MatchNot* $m1$)) = *MatchNot*
MatchAny then
remove-unknowns-generic $(\beta, \alpha) a$ (*MatchNot* $m2$) else
if (*remove-unknowns-generic* $(\beta, \alpha) a$ (*MatchNot* $m2$)) = *MatchNot*
MatchAny then
remove-unknowns-generic $(\beta, \alpha) a$ (*MatchNot* $m1$) else
MatchNot (*MatchAnd* (*MatchNot* (*remove-unknowns-generic* $(\beta, \alpha) a$
(*MatchNot* $m1$))) (*MatchNot* (*remove-unknowns-generic* $(\beta, \alpha) a$ (*MatchNot* $m2$))))))
)

lemma[code-unfold]: *remove-unknowns-generic* γa (*MatchNot* (*MatchAnd* $m1 m2$))

= (let $m1' = \text{remove-unknowns-generic } \gamma a (\text{MatchNot } m1)$; $m2' = \text{remove-unknowns-generic}$
 $\gamma a (\text{MatchNot } m2)$ in
(if $m1' = \text{MatchAny} \vee m2' = \text{MatchAny}$
then *MatchAny*
else
if $m1' = \text{MatchNot MatchAny}$ then $m2'$ else
if $m2' = \text{MatchNot MatchAny}$ then $m1'$
else
MatchNot (*MatchAnd* (*MatchNot* $m1'$) (*MatchNot* $m2'$)))
)

<proof>

lemma *remove-unknowns-generic-simp-3-4-unfolded*: *remove-unknowns-generic* $(\beta,$
 $\alpha) a$ (*Match* A) = (if
 $(\forall p. \text{ternary-ternary-eval} (\text{map-match-tac } \beta p (\text{Match } A)) = \text{TernaryUnknown})$
then
if $(\forall p. \alpha a p)$ then *MatchAny* else if $(\forall p. \neg \alpha a p)$ then *MatchNot MatchAny*
else *Match* A
else (*Match* A))

remove-unknowns-generic (β, α) a (*MatchNot* (*Match* A)) = (if
 $(\forall p. \text{ternary-ternary-eval } (\text{map-match-tac } \beta p (\text{Match } A)) = \text{TernaryUnknown})$
then
if $(\forall p. \alpha a p)$ then *MatchAny* else if $(\forall p. \neg \alpha a p)$ then *MatchNot MatchAny*
else *MatchNot* (*Match* A)
else *MatchNot* (*Match* A)
⟨proof⟩

declare *remove-unknowns-generic.simps*[simp del]

lemmas *remove-unknowns-generic-simps2* = *remove-unknowns-generic.simps*(1)
remove-unknowns-generic.simps(2)
remove-unknowns-generic-simp-3-4-unfolded
remove-unknowns-generic.simps(5) *remove-unknowns-generic.simps*(6)
remove-unknowns-generic.simps(7)

lemma *matches* (β, α) (*remove-unknowns-generic* (β, α) a (*MatchNot* (*Match* A))) $a p$ = *matches* (β, α) (*MatchNot* (*Match* A)) $a p$
⟨proof⟩

lemma *remove-unknowns-generic: matches* γ (*remove-unknowns-generic* $\gamma a m$) a
= *matches* $\gamma m a$
⟨proof⟩

fun *has-unknowns* :: ('a, 'p) *exact-match-tac* \Rightarrow 'a *match-expr* \Rightarrow bool **where**
has-unknowns β (*Match* A) = $(\exists p. \text{ternary-ternary-eval } (\text{map-match-tac } \beta p (\text{Match } A)) = \text{TernaryUnknown})$ |
has-unknowns β (*MatchNot* m) = *has-unknowns* βm |
has-unknowns β *MatchAny* = False |
has-unknowns β (*MatchAnd* $m1 m2$) = (*has-unknowns* $\beta m1 \vee \text{has-unknowns } \beta m2$)

definition *packet-independent- α* :: 'p *unknown-match-tac* \Rightarrow bool **where**
packet-independent- α α = $(\forall a p1 p2. a = \text{Accept} \vee a = \text{Drop} \longrightarrow \alpha a p1 \longleftrightarrow \alpha a p2)$

lemma *packet-independent-unknown-match*: $a = \text{Accept} \vee a = \text{Drop} \implies \text{packet-independent-}\alpha \alpha \implies \neg \text{unknown-not-match-any } \alpha a \longleftrightarrow \text{unknown-match-all } \alpha a$
⟨proof⟩

If for some type the exact matcher returns unknown, then it returns unknown

for all these types

definition *packet-independent-β-unknown* :: ('a, 'packet) exact-match-tac ⇒ bool
where
packet-independent-β-unknown β ≡ ∀ A. (∃ p. β A p ≠ TernaryUnknown) →
(∀ p. β A p ≠ TernaryUnknown)

lemma *remove-unknowns-generic-specification*: a = Accept ∨ a = Drop ⇒ *packet-independent-α*
α ⇒
packet-independent-β-unknown β ⇒
¬ has-unknowns β (remove-unknowns-generic (β, α) a m)
⟨proof⟩

Checking is something matches unconditionally

context

begin

private lemma *no-primitives-no-unknown*: ¬ has-primitive m ⇒ (ternary-ternary-eval
(map-match-tac β p m)) ≠ TernaryUnknown
⟨proof⟩ **lemma** *no-primitives-matchNot*: **assumes** ¬ has-primitive m **shows**
matches γ (MatchNot m) a p ↔ ¬ matches γ m a p
⟨proof⟩

lemma *matcheq-matchAny*: ¬ has-primitive m ⇒ matcheq-matchAny m ↔
matches γ m a p
⟨proof⟩

lemma *matcheq-matchNone*: ¬ has-primitive m ⇒ matcheq-matchNone m ↔
¬ matches γ m a p
⟨proof⟩

lemma *matcheq-matchNone-not-matches*: matcheq-matchNone m ⇒ ¬ matches
γ m a p
⟨proof⟩

end

Lemmas about *MatchNot* in ternary logic.

lemma *matches-MatchNot-no-unknowns*:
assumes ¬ has-unknowns β m
shows matches (β,α) (MatchNot m) a p ↔ ¬ matches (β,α) m a p
⟨proof⟩

lemma *MatchNot-ternary-ternary-eval*: (ternary-ternary-eval (map-match-tac β p
m')) = (ternary-ternary-eval (map-match-tac β p m)) ⇒
matches (β,α) (MatchNot m') a p = matches (β,α) (MatchNot m) a p
⟨proof⟩

For our 'p unknown-match-tacs *in-doubt-allow* and *in-doubt-deny*, when do-

ing an induction over some function that modifies m , we get the *MatchNot* case for free (if we can set arbitrary p). This does not hold for arbitrary ' p unknown-match-tacs.

lemma *matches-induction-case-MatchNot*:

assumes α Drop \neq α Accept **and** packet-independent- α α

and $\forall a. \text{matches } (\beta, \alpha) m' a p = \text{matches } (\beta, \alpha) m a p$

shows $\text{matches } (\beta, \alpha) (\text{MatchNot } m') a p = \text{matches } (\beta, \alpha) (\text{MatchNot } m) a$

p
 $\langle \text{proof} \rangle$

end

theory *Semantics-Ternary*

imports *Matching-Ternary ../Common/List-Misc*

begin

8 Embedded Ternary-Matching Big Step Semantics

8.1 Ternary Semantics (Big Step)

inductive *approximating-bigstep* :: ('a, 'p) match-tac \Rightarrow 'p \Rightarrow 'a rule list \Rightarrow state \Rightarrow state \Rightarrow bool

$\langle \cdot, + \langle \cdot, \cdot \rangle \Rightarrow_{\alpha} \cdot \rangle$ [60,60,20,98,98] 89

for γ **and** p **where**

skip: $\gamma, p \vdash \langle [], t \rangle \Rightarrow_{\alpha} t$ |

accept: $\llbracket \text{matches } \gamma m \text{ Accept } p \rrbracket \Longrightarrow \gamma, p \vdash \langle [\text{Rule } m \text{ Accept}], \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow}$ |

drop: $\llbracket \text{matches } \gamma m \text{ Drop } p \rrbracket \Longrightarrow \gamma, p \vdash \langle [\text{Rule } m \text{ Drop}], \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalDeny}$ |

reject: $\llbracket \text{matches } \gamma m \text{ Reject } p \rrbracket \Longrightarrow \gamma, p \vdash \langle [\text{Rule } m \text{ Reject}], \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalDeny}$ |

log: $\llbracket \text{matches } \gamma m \text{ Log } p \rrbracket \Longrightarrow \gamma, p \vdash \langle [\text{Rule } m \text{ Log}], \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Undecided}$ |

empty: $\llbracket \text{matches } \gamma m \text{ Empty } p \rrbracket \Longrightarrow \gamma, p \vdash \langle [\text{Rule } m \text{ Empty}], \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Undecided}$ |

nomatch: $\llbracket \neg \text{matches } \gamma m a p \rrbracket \Longrightarrow \gamma, p \vdash \langle [\text{Rule } m a], \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Undecided}$ |

decision: $\gamma, p \vdash \langle rs, \text{Decision } X \rangle \Rightarrow_{\alpha} \text{Decision } X$ |

seq: $\llbracket \gamma, p \vdash \langle rs_1, \text{Undecided} \rangle \Rightarrow_{\alpha} t; \gamma, p \vdash \langle rs_2, t \rangle \Rightarrow_{\alpha} t' \rrbracket \Longrightarrow \gamma, p \vdash \langle rs_1 @ rs_2, \text{Undecided} \rangle \Rightarrow_{\alpha} t'$

thm *approximating-bigstep.induct*[of γ p rs s t P]

lemma *approximating-bigstep-induct*[case-names *Skip Allow Deny Log Nomatch Decision Seq*, *induct pred: approximating-bigstep*] : $\gamma, p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t \Longrightarrow$

$(\bigwedge t. P \square t) \implies$
 $(\bigwedge m a. \text{matches } \gamma m a p \implies a = \text{Accept} \implies P [\text{Rule } m a] \text{ Undecided } (\text{Decision FinalAllow})) \implies$
 $(\bigwedge m a. \text{matches } \gamma m a p \implies a = \text{Drop} \vee a = \text{Reject} \implies P [\text{Rule } m a] \text{ Undecided } (\text{Decision FinalDeny})) \implies$
 $(\bigwedge m a. \text{matches } \gamma m a p \implies a = \text{Log} \vee a = \text{Empty} \implies P [\text{Rule } m a] \text{ Undecided Undecided}) \implies$
 $(\bigwedge m a. \neg \text{matches } \gamma m a p \implies P [\text{Rule } m a] \text{ Undecided Undecided}) \implies$
 $(\bigwedge rs X. P rs (\text{Decision } X) (\text{Decision } X)) \implies$
 $(\bigwedge rs rs_1 rs_2 t t'. rs = rs_1 @ rs_2 \implies \gamma, p \vdash \langle rs_1, \text{Undecided} \rangle \Rightarrow_\alpha t \implies P rs_1 \text{ Undecided } t \implies \gamma, p \vdash \langle rs_2, t \rangle \Rightarrow_\alpha t' \implies P rs_2 t t' \implies P rs \text{ Undecided } t') \implies P rs s t$
 $\langle \text{proof} \rangle$

lemma skipD: $\gamma, p \vdash \langle [], s \rangle \Rightarrow_\alpha t \implies s = t$
 $\langle \text{proof} \rangle$

lemma decisionD: $\gamma, p \vdash \langle rs, \text{Decision } X \rangle \Rightarrow_\alpha t \implies t = \text{Decision } X$
 $\langle \text{proof} \rangle$

lemma acceptD: $\gamma, p \vdash \langle [\text{Rule } m \text{ Accept}], \text{Undecided} \rangle \Rightarrow_\alpha t \implies \text{matches } \gamma m \text{ Accept } p \implies t = \text{Decision FinalAllow}$
 $\langle \text{proof} \rangle$

lemma dropD: $\gamma, p \vdash \langle [\text{Rule } m \text{ Drop}], \text{Undecided} \rangle \Rightarrow_\alpha t \implies \text{matches } \gamma m \text{ Drop } p \implies t = \text{Decision FinalDeny}$
 $\langle \text{proof} \rangle$

lemma rejectD: $\gamma, p \vdash \langle [\text{Rule } m \text{ Reject}], \text{Undecided} \rangle \Rightarrow_\alpha t \implies \text{matches } \gamma m \text{ Reject } p \implies t = \text{Decision FinalDeny}$
 $\langle \text{proof} \rangle$

lemma logD: $\gamma, p \vdash \langle [\text{Rule } m \text{ Log}], \text{Undecided} \rangle \Rightarrow_\alpha t \implies t = \text{Undecided}$
 $\langle \text{proof} \rangle$

lemma emptyD: $\gamma, p \vdash \langle [\text{Rule } m \text{ Empty}], \text{Undecided} \rangle \Rightarrow_\alpha t \implies t = \text{Undecided}$
 $\langle \text{proof} \rangle$

lemma nomatchD: $\gamma, p \vdash \langle [\text{Rule } m a], \text{Undecided} \rangle \Rightarrow_\alpha t \implies \neg \text{matches } \gamma m a p \implies t = \text{Undecided}$
 $\langle \text{proof} \rangle$

lemmas approximating-bigstepD = skipD acceptD dropD rejectD logD emptyD nomatchD decisionD

lemma approximating-bigstep-to-undecided: $\gamma, p \vdash \langle rs, s \rangle \Rightarrow_\alpha \text{Undecided} \implies s = \text{Undecided}$
 $\langle \text{proof} \rangle$

lemma *approximating-bigstep-to-decision1*: $\gamma, p \vdash \langle rs, \text{Decision } Y \rangle \Rightarrow_{\alpha} \text{Decision } X$
 $\Rightarrow Y = X$
 ⟨proof⟩

lemma *nomatch-fst*: $\neg \text{matches } \gamma \ m \ a \ p \Rightarrow \gamma, p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t \Rightarrow \gamma, p \vdash \langle \text{Rule}$
 $m \ a \ \# \ rs, s \rangle \Rightarrow_{\alpha} t$
 ⟨proof⟩

lemma *seq'*:
assumes $rs = rs_1 \ @ \ rs_2 \ \gamma, p \vdash \langle rs_1, s \rangle \Rightarrow_{\alpha} t \ \gamma, p \vdash \langle rs_2, t \rangle \Rightarrow_{\alpha} t'$
shows $\gamma, p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t'$
 ⟨proof⟩

lemma *seq-split*:
assumes $\gamma, p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t \ rs = rs_1 \ @ \ rs_2$
obtains t' **where** $\gamma, p \vdash \langle rs_1, s \rangle \Rightarrow_{\alpha} t' \ \gamma, p \vdash \langle rs_2, t' \rangle \Rightarrow_{\alpha} t$
 ⟨proof⟩

lemma *seqE-fst*:
assumes $\gamma, p \vdash \langle r \# rs, s \rangle \Rightarrow_{\alpha} t$
obtains t' **where** $\gamma, p \vdash \langle [r], s \rangle \Rightarrow_{\alpha} t' \ \gamma, p \vdash \langle rs, t' \rangle \Rightarrow_{\alpha} t$
 ⟨proof⟩

lemma *seq-fst*: **assumes** $\gamma, p \vdash \langle [r], s \rangle \Rightarrow_{\alpha} t$ **and** $\gamma, p \vdash \langle rs, t \rangle \Rightarrow_{\alpha} t'$ **shows** $\gamma, p \vdash$
 $\langle r \# rs, s \rangle \Rightarrow_{\alpha} t'$
 ⟨proof⟩

8.2 wf ruleset

A 'a rule list here is well-formed (for a packet) if

- either the rules do not match
- or the action is not *Call*, not *Return*, not *Unknown*

definition *wf-ruleset* :: ('a, 'p) match-tac \Rightarrow 'p \Rightarrow 'a rule list \Rightarrow bool **where**
 $wf\text{-ruleset } \gamma \ p \ rs \equiv \forall r \in \text{set } rs.$
 $(\neg \text{matches } \gamma \ (\text{get-match } r) \ (\text{get-action } r) \ p) \vee$
 $(\neg(\exists \text{chain. } \text{get-action } r = \text{Call chain}) \wedge \text{get-action } r \neq \text{Return} \wedge \neg(\exists \text{chain.}$
 $\text{get-action } r = \text{Goto chain}) \wedge \text{get-action } r \neq \text{Unknown})$

lemma *wf-ruleset-append*: $wf\text{-ruleset } \gamma \ p \ (rs1 \ @ \ rs2) \longleftrightarrow wf\text{-ruleset } \gamma \ p \ rs1 \ \wedge$
 $wf\text{-ruleset } \gamma \ p \ rs2$
 ⟨proof⟩

lemma *wf-rulesetD*: **assumes** $wf\text{-ruleset } \gamma \ p \ (r \ \# \ rs)$ **shows** $wf\text{-ruleset } \gamma \ p \ [r]$
and $wf\text{-ruleset } \gamma \ p \ rs$
 ⟨proof⟩

lemma *wf-ruleset-fst*: $wf\text{-ruleset } \gamma p (Rule\ m\ a\ \# rs) \longleftrightarrow wf\text{-ruleset } \gamma p [Rule\ m\ a] \wedge wf\text{-ruleset } \gamma p rs$
 ⟨proof⟩

lemma *wf-ruleset-stripfst*: $wf\text{-ruleset } \gamma p (r\ \# rs) \implies wf\text{-ruleset } \gamma p (rs)$
 ⟨proof⟩

lemma *wf-ruleset-rest*: $wf\text{-ruleset } \gamma p (Rule\ m\ a\ \# rs) \implies wf\text{-ruleset } \gamma p [Rule\ m\ a]$
 ⟨proof⟩

8.3 Ternary Semantics (Function)

fun *approximating-bigstep-fun* :: ('a, 'p) *match-tac* \Rightarrow 'p \Rightarrow 'a *rule list* \Rightarrow *state* \Rightarrow *state* **where**

approximating-bigstep-fun $\gamma p [] s = s$ |
approximating-bigstep-fun $\gamma p rs (Decision\ X) = (Decision\ X)$ |
approximating-bigstep-fun $\gamma p ((Rule\ m\ a)\ \# rs)\ Undecided = (if$
 $\neg\ matches\ \gamma\ m\ a\ p$
 then
 approximating-bigstep-fun $\gamma p rs\ Undecided$
 else
 case a of *Accept* $\Rightarrow Decision\ FinalAllow$
 | *Drop* $\Rightarrow Decision\ FinalDeny$
 | *Reject* $\Rightarrow Decision\ FinalDeny$
 | *Log* $\Rightarrow approximating\text{-bigstep-fun } \gamma p rs\ Undecided$
 | *Empty* $\Rightarrow approximating\text{-bigstep-fun } \gamma p rs\ Undecided$
 — unhandled cases
)

lemma *approximating-bigstep-fun-induct*[*case-names Empty Decision Nomatch Match*]:

$(\bigwedge \gamma p s. P\ \gamma p [] s) \implies$
 $(\bigwedge \gamma p r rs X. P\ \gamma p (r\ \# rs) (Decision\ X)) \implies$
 $(\bigwedge \gamma p m a rs.$
 $\neg\ matches\ \gamma\ m\ a\ p \implies P\ \gamma p rs\ Undecided \implies P\ \gamma p (Rule\ m\ a\ \# rs)$
Undecided) \implies
 $(\bigwedge \gamma p m a rs.$
 $matches\ \gamma\ m\ a\ p \implies (a = Log \implies P\ \gamma p rs\ Undecided) \implies (a = Empty \implies$
 $P\ \gamma p rs\ Undecided) \implies P\ \gamma p (Rule\ m\ a\ \# rs)\ Undecided) \implies$
 $P\ \gamma p rs\ s$
 ⟨proof⟩

lemma *Decision-approximating-bigstep-fun*: $approximating\text{-bigstep-fun } \gamma p rs (Decision\ X) = Decision\ X$
 ⟨proof⟩

lemma *approximating-bigstep-fun-induct-wf*[*case-names Empty Decision Nomatch MatchAccept MatchDrop MatchReject MatchLog MatchEmpty, consumes 1*]:

$$\begin{aligned}
& \text{wf-ruleset } \gamma \text{ p rs} \implies \\
& (\bigwedge \gamma \text{ p s. } P \gamma \text{ p } \llbracket s \rrbracket) \implies \\
& (\bigwedge \gamma \text{ p r rs X. } P \gamma \text{ p } (r \# \text{rs}) \text{ (Decision X)}) \implies \\
& (\bigwedge \gamma \text{ p m a rs.} \\
& \quad \neg \text{matches } \gamma \text{ m a p} \implies P \gamma \text{ p rs Undecided} \implies P \gamma \text{ p (Rule m a \# rs)} \\
& \quad \text{Undecided}) \implies \\
& (\bigwedge \gamma \text{ p m a rs.} \\
& \quad \text{matches } \gamma \text{ m a p} \implies a = \text{Accept} \implies P \gamma \text{ p (Rule m a \# rs) Undecided}) \implies \\
& (\bigwedge \gamma \text{ p m a rs.} \\
& \quad \text{matches } \gamma \text{ m a p} \implies a = \text{Drop} \implies P \gamma \text{ p (Rule m a \# rs) Undecided}) \implies \\
& (\bigwedge \gamma \text{ p m a rs.} \\
& \quad \text{matches } \gamma \text{ m a p} \implies a = \text{Reject} \implies P \gamma \text{ p (Rule m a \# rs) Undecided}) \implies \\
& (\bigwedge \gamma \text{ p m a rs.} \\
& \quad \text{matches } \gamma \text{ m a p} \implies a = \text{Log} \implies P \gamma \text{ p rs Undecided} \implies P \gamma \text{ p (Rule m a} \\
& \quad \# \text{rs) Undecided}) \implies \\
& (\bigwedge \gamma \text{ p m a rs.} \\
& \quad \text{matches } \gamma \text{ m a p} \implies a = \text{Empty} \implies P \gamma \text{ p rs Undecided} \implies P \gamma \text{ p (Rule m} \\
& \quad \text{a \# rs) Undecided}) \implies \\
& P \gamma \text{ p rs s} \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *just-show-all-approximating-bigstep-fun-equalities-with-start-Undecided*[*case-names Undecided*]:

$$\begin{aligned}
& \text{assumes } s = \text{Undecided} \implies \text{approximating-bigstep-fun } \gamma \text{ p rs1 s} = \text{approximating-bigstep-fun } \gamma \text{ p rs2 s} \\
& \text{shows } \text{approximating-bigstep-fun } \gamma \text{ p rs1 s} = \text{approximating-bigstep-fun } \gamma \text{ p rs2 s} \\
& \langle \text{proof} \rangle
\end{aligned}$$

8.3.1 Append, Prepend, Postpend, Composition

lemma *approximating-bigstep-fun-seq-wf*: $\llbracket \text{wf-ruleset } \gamma \text{ p rs}_1 \rrbracket \implies$
 $\text{approximating-bigstep-fun } \gamma \text{ p } (rs_1 @ rs_2) s = \text{approximating-bigstep-fun } \gamma \text{ p } rs_2$
 $(\text{approximating-bigstep-fun } \gamma \text{ p } rs_1 s)$
 $\langle \text{proof} \rangle$

The state transitions from *Undecided* to *Undecided* if all intermediate states are *Undecided*

lemma *approximating-bigstep-fun-seq-Undecided-wf*: $\llbracket \text{wf-ruleset } \gamma \text{ p } (rs_1 @ rs_2) \rrbracket$
 \implies
 $\text{approximating-bigstep-fun } \gamma \text{ p } (rs_1 @ rs_2) \text{ Undecided} = \text{Undecided} \iff$
 $\text{approximating-bigstep-fun } \gamma \text{ p } rs_1 \text{ Undecided} = \text{Undecided} \wedge \text{approximating-bigstep-fun } \gamma \text{ p } rs_2$
 $\text{ Undecided} = \text{Undecided}$
 $\langle \text{proof} \rangle$

lemma *approximating-bigstep-fun-seq-Undecided-t-wf*: $\llbracket \text{wf-ruleset } \gamma \text{ p } (rs_1 @ rs_2) \rrbracket$

\implies
 $\text{approximating-bigstep-fun } \gamma \ p \ (rs1 @ rs2) \ \text{Undecided} = t \iff$
 $\text{approximating-bigstep-fun } \gamma \ p \ rs1 \ \text{Undecided} = \text{Undecided} \wedge \text{approximating-bigstep-fun}$
 $\gamma \ p \ rs2 \ \text{Undecided} = t \vee$
 $\text{approximating-bigstep-fun } \gamma \ p \ rs1 \ \text{Undecided} = t \wedge t \neq \text{Undecided}$
 $\langle \text{proof} \rangle$

lemma *approximating-bigstep-fun-wf-postpend*: $\text{wf-ruleset } \gamma \ p \ rsA \implies \text{wf-ruleset}$
 $\gamma \ p \ rsB \implies$
 $\text{approximating-bigstep-fun } \gamma \ p \ rsA \ s = \text{approximating-bigstep-fun } \gamma \ p \ rsB \ s$
 \implies
 $\text{approximating-bigstep-fun } \gamma \ p \ (rsA @ rsC) \ s = \text{approximating-bigstep-fun } \gamma \ p$
 $(rsB @ rsC) \ s$
 $\langle \text{proof} \rangle$

lemma *approximating-bigstep-fun-singleton-prepend*:
assumes $\text{approximating-bigstep-fun } \gamma \ p \ rsB \ s = \text{approximating-bigstep-fun } \gamma \ p$
 $rsC \ s$
shows $\text{approximating-bigstep-fun } \gamma \ p \ (r \# rsB) \ s = \text{approximating-bigstep-fun } \gamma$
 $p \ (r \# rsC) \ s$
 $\langle \text{proof} \rangle$

8.4 Equality with $\gamma, p \vdash \langle rs, s \rangle \Rightarrow_\alpha t$ semantics

lemma *approximating-bigstep-wf*: $\gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_\alpha \text{Undecided} \implies \text{wf-ruleset}$
 $\gamma \ p \ rs$
 $\langle \text{proof} \rangle$

only valid actions appear in this ruleset

definition *good-ruleset* :: 'a rule list \Rightarrow bool **where**
 $\text{good-ruleset } rs \equiv \forall r \in \text{set } rs. (\neg(\exists \text{chain. } \text{get-action } r = \text{Call chain}) \wedge \text{get-action}$
 $r \neq \text{Return} \wedge \neg(\exists \text{chain. } \text{get-action } r = \text{Goto chain}) \wedge \text{get-action } r \neq \text{Unknown})$

lemma[code-unfold]: $\text{good-ruleset } rs = (\forall r \in \text{set } rs. (\text{case } \text{get-action } r \text{ of } \text{Call chain}$
 $\Rightarrow \text{False} \mid \text{Return} \Rightarrow \text{False} \mid \text{Goto chain} \Rightarrow \text{False} \mid \text{Unknown} \Rightarrow \text{False} \mid - \Rightarrow \text{True}))$
 $\langle \text{proof} \rangle$

lemma *good-ruleset-alt*: $\text{good-ruleset } rs = (\forall r \in \text{set } rs. \text{get-action } r = \text{Accept} \vee$
 $\text{get-action } r = \text{Drop} \vee$
 $\text{get-action } r = \text{Reject} \vee \text{get-action } r = \text{Log}$
 $\vee \text{get-action } r = \text{Empty})$
 $\langle \text{proof} \rangle$

lemma *good-ruleset-append*: $\text{good-ruleset } (rs_1 @ rs_2) \iff \text{good-ruleset } rs_1 \wedge$
 $\text{good-ruleset } rs_2$

$\langle \text{proof} \rangle$

lemma *good-ruleset-fst*: $\text{good-ruleset } (r\#rs) \implies \text{good-ruleset } [r]$
 $\langle \text{proof} \rangle$

lemma *good-ruleset-tail*: $\text{good-ruleset } (r\#rs) \implies \text{good-ruleset } rs$
 $\langle \text{proof} \rangle$

good-ruleset is stricter than *wf-ruleset*. It can be easily checked with running code!

lemma *good-imp-wf-ruleset*: $\text{good-ruleset } rs \implies \text{wf-ruleset } \gamma \ p \ rs$ $\langle \text{proof} \rangle$

lemma *simple-imp-good-ruleset*: $\text{simple-ruleset } rs \implies \text{good-ruleset } rs$
 $\langle \text{proof} \rangle$

lemma *approximating-bigstep-fun-seq-antics*: $\llbracket \gamma, p \vdash \langle rs_1, s \rangle \Rightarrow_\alpha t \rrbracket \implies$
 $\text{approximating-bigstep-fun } \gamma \ p \ (rs_1 \ @ \ rs_2) \ s = \text{approximating-bigstep-fun } \gamma \ p$
 $rs_2 \ t$
 $\langle \text{proof} \rangle$

lemma *approximating-semantics-imp-fun*: $\gamma, p \vdash \langle rs, s \rangle \Rightarrow_\alpha t \implies \text{approximating-bigstep-fun}$
 $\gamma \ p \ rs \ s = t$
 $\langle \text{proof} \rangle$

lemma *approximating-fun-imp-semantics*: **assumes** $\text{wf-ruleset } \gamma \ p \ rs$
shows $\text{approximating-bigstep-fun } \gamma \ p \ rs \ s = t \implies \gamma, p \vdash \langle rs, s \rangle \Rightarrow_\alpha t$
 $\langle \text{proof} \rangle$

Henceforth, we will use the *approximating-bigstep-fun* semantics, because they are easier. We show that they are equal.

theorem *approximating-semantics-iff-fun*: $\text{wf-ruleset } \gamma \ p \ rs \implies$
 $\gamma, p \vdash \langle rs, s \rangle \Rightarrow_\alpha t \iff \text{approximating-bigstep-fun } \gamma \ p \ rs \ s = t$
 $\langle \text{proof} \rangle$

corollary *approximating-semantics-iff-fun-good-ruleset*: $\text{good-ruleset } rs \implies$
 $\gamma, p \vdash \langle rs, s \rangle \Rightarrow_\alpha t \iff \text{approximating-bigstep-fun } \gamma \ p \ rs \ s = t$
 $\langle \text{proof} \rangle$

lemma *approximating-bigstep-deterministic*: $\llbracket \gamma, p \vdash \langle rs, s \rangle \Rightarrow_\alpha t; \gamma, p \vdash \langle rs, s \rangle \Rightarrow_\alpha$
 $t' \rrbracket \implies t = t'$
 $\langle \text{proof} \rangle$

lemma *rm-LogEmpty-fun-semantics*:
 $\text{approximating-bigstep-fun } \gamma \ p \ (\text{rm-LogEmpty } rs) \ s = \text{approximating-bigstep-fun}$
 $\gamma \ p \ rs \ s$
 $\langle \text{proof} \rangle$

lemma $\gamma, p \vdash \langle \text{rm-LogEmpty } rs, s \rangle \Rightarrow_{\alpha} t \iff \gamma, p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t$
 ⟨proof⟩

lemma *rm-LogEmpty-simple-but-Reject*:
good-ruleset $rs \implies \forall r \in \text{set } (rm\text{-LogEmpty } rs). \text{get-action } r = \text{Accept} \vee \text{get-action } r = \text{Reject} \vee \text{get-action } r = \text{Drop}$
 ⟨proof⟩

lemma *rw-Reject-fun-semantics*:
wf-unknown-match-tac $\alpha \implies$
 $(\text{approximating-bigstep-fun } (\beta, \alpha) p (rw\text{-Reject } rs) s = \text{approximating-bigstep-fun } (\beta, \alpha) p rs s)$
 ⟨proof⟩

lemma *rmLogEmpty-rwReject-good-to-simple*: *good-ruleset* $rs \implies \text{simple-ruleset } (rw\text{-Reject } (rm\text{-LogEmpty } rs))$
 ⟨proof⟩

8.5 Matching

lemma *optimize-matches-option-generic*:
assumes $\forall r \in \text{set } rs. P (\text{get-match } r) (\text{get-action } r)$
and $(\bigwedge m m' a. P m a \implies f m = \text{Some } m' \implies \text{matches } \gamma m' a p = \text{matches } \gamma m a p)$
and $(\bigwedge m a. P m a \implies f m = \text{None} \implies \neg \text{matches } \gamma m a p)$
shows $\text{approximating-bigstep-fun } \gamma p (\text{optimize-matches-option } f rs) s = \text{approximating-bigstep-fun } \gamma p rs s$
 ⟨proof⟩

lemma *optimize-matches-generic*: $\forall r \in \text{set } rs. P (\text{get-match } r) (\text{get-action } r) \implies$
 $(\bigwedge m a. P m a \implies \text{matches } \gamma (f m) a p = \text{matches } \gamma m a p) \implies$
 $\text{approximating-bigstep-fun } \gamma p (\text{optimize-matches } f rs) s = \text{approximating-bigstep-fun } \gamma p rs s$
 ⟨proof⟩

lemma *optimize-matches-matches-fst*: $\text{matches } \gamma (f m) a p \implies \text{optimize-matches } f (\text{Rule } m a \# rs) = (\text{Rule } (f m) a) \# \text{optimize-matches } f rs$
 ⟨proof⟩

lemma *optimize-matches*: $\forall m a. \text{matches } \gamma (f m) a p = \text{matches } \gamma m a p \implies \text{ap-}$

approximating-bigstep-fun γ *p* (*optimize-matches* *f* *rs*) *s* = *approximating-bigstep-fun*
 γ *p* *rs* *s*
 ⟨*proof*⟩

lemma *optimize-matches-opt-MatchAny-match-expr*: *approximating-bigstep-fun* γ
p (*optimize-matches* *opt-MatchAny-match-expr* *rs*) *s* = *approximating-bigstep-fun*
 γ *p* *rs* *s*
 ⟨*proof*⟩

lemma *optimize-matches-a*: $\forall a m. \text{matches } \gamma m a = \text{matches } \gamma (f a m) a \implies$ *ap-*
proximating-bigstep-fun γ *p* (*optimize-matches-a* *f* *rs*) *s* = *approximating-bigstep-fun*
 γ *p* *rs* *s*
 ⟨*proof*⟩

lemma *optimize-matches-a-simplers*:
assumes *simple-ruleset* *rs* **and** $\forall a m. a = \text{Accept} \vee a = \text{Drop} \longrightarrow \text{matches } \gamma (f$
 $a m) a = \text{matches } \gamma m a$
shows *approximating-bigstep-fun* γ *p* (*optimize-matches-a* *f* *rs*) *s* = *approximat-*
ing-bigstep-fun γ *p* *rs* *s*
 ⟨*proof*⟩

lemma *not-matches-removeAll*: $\neg \text{matches } \gamma m a p \implies$
approximating-bigstep-fun γ *p* (*removeAll* (*Rule* *m* *a*) *rs*) *Undecided* = *approxim-*
ating-bigstep-fun γ *p* *rs* *Undecided*
 ⟨*proof*⟩

end
theory *Datatype-Selectors*
imports *Main*
begin

Running Example: *datatype-new ipt-rule-match* = *is-Src*: *Src* (*src-range*: *ipt-ipt-range*)

A discriminator *disc* tells whether a value is of a certain constructor. Ex-
 ample: *is-Src*

A selector *sel* select the inner value. Example: *src-range*

A constructor *C* constructs a value Example: *Src*

The are well-formed if the belong together.

fun *wf-disc-sel* :: $((a \Rightarrow \text{bool}) \times (a \Rightarrow b)) \Rightarrow (b \Rightarrow a) \Rightarrow \text{bool}$ **where**
wf-disc-sel (*disc*, *sel*) *C* $\longleftrightarrow (\forall a. \text{disc } a \longrightarrow C (\text{sel } a) = a) \wedge (\forall a. \text{disc } a \longrightarrow$
 $\text{disc } (C a) = a)$

```

declare wf-disc-sel.simps[simp del]

end
theory IpAddresses
imports IP-Addresses.IP-Address-toString
         IP-Addresses.CIDR-Split
         ../Common/WordInterval-Lists
begin

```

— Misc

```

lemma ipset-from-cidr (ipv4addr-of-dotdecimal (0, 0, 0, 0)) 33 = {0}
  <proof>

```

```

definition all-but-those-ips :: ('i::len word × nat) list ⇒ ('i word × nat) list
where
  all-but-those-ips cidrips = cidr-split (wordinterval-invert (l2wi (map ipcidr-to-interval
cidrips)))

```

```

lemma all-but-those-ips:
  ipcidr-union-set (set (all-but-those-ips cidrips)) =
    UNIV − (⋃ (ip,n) ∈ set cidrips. ipset-from-cidr ip n)
  <proof>

```

9 IPv4 Addresses

9.1 IPv4 Addresses in IPTables Notation (how we parse it)

```

context
  notes [[typedef-overloaded]]
begin
  datatype 'i ipt-irange =
    — Singleton IP Address
    | IpAddr 'i::len word

    — CIDR notation: addr/xx
    | IpAddrNetmask 'i word nat

    — -m iprange -src-range a.b.c.d-e.f.g.h
    | IpAddrRange 'i word 'i word
end

```

```

fun ipt-iprange-to-set :: 'i::len ipt-iprange  $\Rightarrow$  'i word set where
  ipt-iprange-to-set (IpAddrNetmask base m) = ipset-from-cidr base m |
  ipt-iprange-to-set (IpAddr ip) = { ip } |
  ipt-iprange-to-set (IpAddrRange ip1 ip2) = { ip1 .. ip2 }

```

ipt-iprange-to-set can only represent an empty set if it is an empty range.

```

lemma ipt-iprange-to-set-nonempty: ipt-iprange-to-set ip = {}  $\longleftrightarrow$ 
  ( $\exists$  ip1 ip2. ip = IpAddrRange ip1 ip2  $\wedge$  ip1 > ip2)
  <proof>

```

maybe this is necessary as code equation?

```

context
  includes bit-operations-syntax
begin

```

```

lemma element-ipt-iprange-to-set[code-unfold]: (addr::'i::len word)  $\in$  ipt-iprange-to-set
X = (
  case X of (IpAddrNetmask pre len)  $\Rightarrow$ 
    (pre AND ((mask len) <<< (len-of (TYPE('i)) - len)))  $\leq$  addr  $\wedge$ 
    addr  $\leq$  pre OR (mask (len-of (TYPE('i)) - len))
  | IpAddr ip  $\Rightarrow$  (addr = ip)
  | IpAddrRange ip1 ip2  $\Rightarrow$  ip1  $\leq$  addr  $\wedge$  ip2  $\geq$  addr)
  <proof>

```

end

```

lemma ipt-iprange-to-set-uncurry-IpAddrNetmask:
  ipt-iprange-to-set (uncurry IpAddrNetmask a) = uncurry ipset-from-cidr a
  <proof>

```

IP address ranges to (*start*, *end*) notation

```

fun ipt-iprange-to-interval :: 'i::len ipt-iprange  $\Rightarrow$  ('i word  $\times$  'i word) where
  ipt-iprange-to-interval (IpAddr addr) = (addr, addr) |
  ipt-iprange-to-interval (IpAddrNetmask pre len) = ipcidr-to-interval (pre, len) |
  ipt-iprange-to-interval (IpAddrRange ip1 ip2) = (ip1, ip2)

```

```

lemma ipt-iprange-to-interval: ipt-iprange-to-interval ip = (s,e)  $\Longrightarrow$  {s .. e} =
ipt-iprange-to-set ip
  <proof>

```

A list of IP address ranges to a '*i* wordinterval'. The nice thing is: the usual set operations are defined on this type. We can use the existing function *l2wi-intersect* if we want the intersection of the supplied list

```

lemma wordinterval-to-set (l2wi-intersect (map ipt-iprange-to-interval ips)) =
  ( $\bigcap$  ip  $\in$  set ips. ipt-iprange-to-set ip)
  <proof>

```

We can use *l2wi* if we want the union of the supplied list

lemma *wordinterval-to-set* (*l2wi* (*map ipt-iprange-to-interval ips*)) = ($\bigcup ip \in set\ ips.\ ipt-iprange-to-set\ ip$)
 ⟨*proof*⟩

A list of (negated) IP address to a 'i *wordinterval*.

definition *ipt-iprange-negation-type-to-br-intersect* ::
 'i::len *ipt-iprange negation-type list* \Rightarrow 'i *wordinterval* **where**
ipt-iprange-negation-type-to-br-intersect l = *l2wi-negation-type-intersect* (*NegPos-map*
ipt-iprange-to-interval l)

lemma *ipt-iprange-negation-type-to-br-intersect: wordinterval-to-set* (*ipt-iprange-negation-type-to-br-intersect*
 l) =
 ($\bigcap ip \in set\ (getPos\ l).\ ipt-iprange-to-set\ ip$) - ($\bigcup ip \in set\ (getNeg\ l).\$
ipt-iprange-to-set ip)
 ⟨*proof*⟩

The 'i *wordinterval* can be translated back into a list of IP ranges. If a list of intervals is enough, we can use *wi2l*. If we need it in 'i *ipt-iprange*, we can use this function.

definition *wi-2-cidr-ipt-iprange-list* :: 'i::len *wordinterval* \Rightarrow 'i *ipt-iprange list*
where
wi-2-cidr-ipt-iprange-list r = *map* (*uncurry IpAddrNetmask*) (*cidr-split* r)

lemma *wi-2-cidr-ipt-iprange-list:*
 ($\bigcup ip \in set\ (wi-2-cidr-ipt-iprange-list\ r).\ ipt-iprange-to-set\ ip$) = *wordinter-*
val-to-set r
 ⟨*proof*⟩

For example, this allows the following transformation

definition *ipt-iprange-compress* :: 'i::len *ipt-iprange negation-type list* \Rightarrow 'i *ipt-iprange*
list **where**
ipt-iprange-compress = *wi-2-cidr-ipt-iprange-list* \circ *ipt-iprange-negation-type-to-br-intersect*

lemma *ipt-iprange-compress:* ($\bigcup ip \in set\ (ipt-iprange-compress\ l).\ ipt-iprange-to-set$
ip) =
 ($\bigcap ip \in set\ (getPos\ l).\ ipt-iprange-to-set\ ip$) - ($\bigcup ip \in set\ (getNeg\ l).\$
ipt-iprange-to-set ip)
 ⟨*proof*⟩

definition *normalized-cidr-ip* :: 'i::len *ipt-iprange* \Rightarrow *bool* **where**
normalized-cidr-ip ip \equiv *case ip of IpAddrNetmask - - \Rightarrow True | - \Rightarrow False*

lemma *wi-2-cidr-ipt-iprange-list-normalized-IpAddrNetmask:*
 $\forall a' \in set\ (wi-2-cidr-ipt-iprange-list\ as).\ normalized-cidr-ip\ a'$
 ⟨*proof*⟩

lemma *ipt-iprange-compress-normalized-IpAddrNetmask:*
 $\forall a' \in set\ (ipt-iprange-compress\ as).\ normalized-cidr-ip\ a'$

<proof>

definition *ipt-iprange-to-cidr* :: 'i::len ipt-iprange ⇒ ('i word × nat) list **where**
ipt-iprange-to-cidr ips = *cidr-split (iprange-interval (ipt-iprange-to-interval ips))*

lemma *ipt-ipvange-to-cidr: ipcidr-union-set (set (ipt-iprange-to-cidr ips)) = (ipt-iprange-to-set ips)*
<proof>

definition *interval-to-wi-to-ipt-iprange* :: 'i::len word ⇒ 'i word ⇒ 'i ipt-iprange **where**
where
interval-to-wi-to-ipt-iprange s e ≡
 if *s = e*
 then *IpAddr s*
 else case *cidr-split (WordInterval s e) of [(ip,nmask)] ⇒ IpAddrNetmask ip nmask*
 | - ⇒ *IpAddrRange s e*

lemma *interval-to-wi-to-ipt-ipv4range: ipt-iprange-to-set (interval-to-wi-to-ipt-iprange s e) = {s..e}*
<proof>

fun *wi-to-ipt-iprange* :: 'i::len wordinterval ⇒ 'i ipt-iprange list **where**
wi-to-ipt-iprange (WordInterval s e) = (if *s > e* then [] else
 [*interval-to-wi-to-ipt-iprange s e*] |
wi-to-ipt-iprange (RangeUnion a b) = *wi-to-ipt-iprange a @ wi-to-ipt-iprange b*

lemma *wi-to-ipt-ipv4range: ⋃ (set (map ipt-iprange-to-set (wi-to-ipt-iprange wi)))*
= wordinterval-to-set wi
<proof>

end
theory *L4-Protocol-Flags*
imports *Simple-Firewall.L4-Protocol*
begin

10 Matching TCP Flags

datatype *ipt-tcp-flags* = *TCP-Flags tcp-flag set* — mask
tcp-flag set — comp

definition *ipt-tcp-syn* :: *ipt-tcp-flags* **where**

$ipt-tcp-syn \equiv TCP-Flags \{TCP-SYN, TCP-RST, TCP-ACK, TCP-FIN\} \{TCP-SYN\}$

fun *match-tcp-flags* :: *ipt-tcp-flags* \Rightarrow *tcp-flag set* \Rightarrow *bool* **where**
match-tcp-flags (*TCP-Flags* *fmask* *c*) *flags* \longleftrightarrow (*flags* \cap *fmask*) = *c*

lemma *match-tcp-flags ipt-tcp-syn* {*TCP-SYN*, *TCP-URG*, *TCP-PSH*} *<proof>*

lemma *match-tcp-flags-nomatch*: $\neg c \subseteq fmask \implies \neg match-tcp-flags$ (*TCP-Flags* *fmask* *c*) *pkt* *<proof>*

definition *ipt-tcp-flags-NoMatch* :: *ipt-tcp-flags* **where**

ipt-tcp-flags-NoMatch $\equiv TCP-Flags \{\} \{TCP-SYN\}$

lemma *ipt-tcp-flags-NoMatch*: $\neg match-tcp-flags$ *ipt-tcp-flags-NoMatch* *pkt* *<proof>*

definition *ipt-tcp-flags-Any* :: *ipt-tcp-flags* **where**

ipt-tcp-flags-Any $\equiv TCP-Flags \{\} \{\}$

lemma *ipt-tcp-flags-Any*: *match-tcp-flags* *ipt-tcp-flags-Any* *pkt* *<proof>*

lemma *ipt-tcp-flags-Any-isUNIV*: $fmask = \{\} \wedge c = \{\} \longleftrightarrow (\forall pkt. match-tcp-flags$ (*TCP-Flags* *fmask* *c*) *pkt*) *<proof>*

fun *match-tcp-flags-conjunct* :: *ipt-tcp-flags* \Rightarrow *ipt-tcp-flags* \Rightarrow *ipt-tcp-flags* **where**
match-tcp-flags-conjunct (*TCP-Flags* *fmask1* *c1*) (*TCP-Flags* *fmask2* *c2*) = (
 if $c1 \subseteq fmask1 \wedge c2 \subseteq fmask2 \wedge fmask1 \cap fmask2 \cap c1 = fmask1 \cap$
fmask2 $\cap c2$
 then (*TCP-Flags* (*fmask1* \cup *fmask2*) (*c1* \cup *c2*))
 else *ipt-tcp-flags-NoMatch*)

lemma *match-tcp-flags-conjunct*: *match-tcp-flags* (*match-tcp-flags-conjunct* *f1* *f2*)
pkt $\longleftrightarrow match-tcp-flags$ *f1* *pkt* $\wedge match-tcp-flags$ *f2* *pkt*
<proof>

declare *match-tcp-flags-conjunct.simps*[*simp del*]

Same as *match-tcp-flags-conjunct*, but returns *None* if result cannot match anyway

definition *match-tcp-flags-conjunct-option* :: *ipt-tcp-flags* \Rightarrow *ipt-tcp-flags* \Rightarrow *ipt-tcp-flags*
option **where**

match-tcp-flags-conjunct-option *f1* *f2* = (case *match-tcp-flags-conjunct* *f1* *f2* of
 (*TCP-Flags* *fmask* *c*) \Rightarrow if $c \subseteq fmask$ then *Some* (*TCP-Flags* *fmask* *c*) else *None*)

lemma *match-tcp-flags-conjunct-option ipt-tcp-syn* (*TCP-Flags* {*TCP-RST*, *TCP-ACK*}
 {*TCP-RST*}) = *None* *<proof>*

lemma *match-tcp-flags-conjunct-option-Some*: *match-tcp-flags-conjunct-option* *f1*
f2 = *Some* *f3* \implies
match-tcp-flags *f1* *pkt* $\wedge match-tcp-flags$ *f2* *pkt* $\longleftrightarrow match-tcp-flags$ *f3* *pkt*
<proof>

lemma *match-tcp-flags-conjunct-option-None*: *match-tcp-flags-conjunct-option* *f1*

$f2 = \text{None} \implies$
 $\neg(\text{match-tcp-flags } f1 \text{ pkt} \wedge \text{match-tcp-flags } f2 \text{ pkt})$
 $\langle \text{proof} \rangle$

lemma *match-tcp-flags-conjunct-option*: (case *match-tcp-flags-conjunct-option* $f1$ $f2$ of $\text{None} \implies \text{False} \mid \text{Some } f3 \implies \text{match-tcp-flags } f3 \text{ pkt}$) $\longleftrightarrow \text{match-tcp-flags } f1 \text{ pkt} \wedge \text{match-tcp-flags } f2 \text{ pkt}$
 $\langle \text{proof} \rangle$

fun *ipt-tcp-flags-equal* :: *ipt-tcp-flags* \Rightarrow *ipt-tcp-flags* \Rightarrow *bool* **where**
ipt-tcp-flags-equal (*TCP-Flags* $fmask1$ $c1$) (*TCP-Flags* $fmask2$ $c2$) = (
 if $c1 \subseteq fmask1 \wedge c2 \subseteq fmask2$
 then $c1 = c2 \wedge fmask1 = fmask2$
 else $(\neg c1 \subseteq fmask1) \wedge (\neg c2 \subseteq fmask2)$)

context
begin

private lemma *funny-set-falg-fmask-helper*: $c2 \subseteq fmask2 \implies (c1 = c2 \wedge fmask1 = fmask2) = (\forall \text{pkt}. (\text{pkt} \cap fmask1 = c1) = (\text{pkt} \cap fmask2 = c2))$
 $\langle \text{proof} \rangle$

lemma *ipt-tcp-flags-equal*: *ipt-tcp-flags-equal* $f1$ $f2 \longleftrightarrow (\forall \text{pkt}. \text{match-tcp-flags } f1 \text{ pkt} = \text{match-tcp-flags } f2 \text{ pkt})$
 $\langle \text{proof} \rangle$

end
declare *ipt-tcp-flags-equal.simps*[*simp del*]

end

theory *Ports*

imports

HOL-Library.Word
 \dots /*Common/WordInterval-Lists*
L4-Protocol-Flags

begin

11 Ports (layer 4)

E.g. source and destination ports for TCP/UDP

list of (start, end) port ranges

type-synonym *raw-ports* = (16 *word* \times 16 *word*) *list*

fun *ports-to-set* :: *raw-ports* \Rightarrow (16 *word*) *set* **where**
ports-to-set [] = {} |
ports-to-set ((s, e)# ps) = { $s..e$ } \cup *ports-to-set* ps

lemma *ports-to-set*: *ports-to-set* $pts = \bigcup \{ \{s..e\} \mid s \ e \ . \ (s, e) \in \text{set } pts \}$
 $\langle \text{proof} \rangle$

We can reuse the wordinterval theory to reason about ports

lemma *ports-to-set-wordinterval*: $ports\text{-}to\text{-}set\ ps = wordinterval\text{-}to\text{-}set\ (l2wi\ ps)$
<proof>

inverting a raw listing of ports

definition *raw-ports-invert* :: $raw\text{-}ports \Rightarrow raw\text{-}ports$ **where**
 $raw\text{-}ports\text{-}invert\ ps = wi2l\ (wordinterval\text{-}invert\ (l2wi\ ps))$

lemma *raw-ports-invert*: $ports\text{-}to\text{-}set\ (raw\text{-}ports\text{-}invert\ ps) = \neg\ ports\text{-}to\text{-}set\ ps$
<proof>

A port always belongs to a protocol! We must not lose this information. You should never use *raw-ports* directly

datatype *ipt-l4-ports* = $L4Ports\ primitive\text{-}protocol\ raw\text{-}ports$

end

theory *Conntrack-State*

imports *../Common/Negation-Type Simple-Firewall.Lib-Enum-toString*

begin

datatype *ctstate* = $CT\text{-}New \mid CT\text{-}Established \mid CT\text{-}Related \mid CT\text{-}Untracked \mid CT\text{-}Invalid$

The state associated with a packet can be added as a tag to the packet. See *../Semantics_Stateful.thy*.

fun *match-ctstate* :: $ctstate\ set \Rightarrow ctstate \Rightarrow bool$ **where**
 $match\text{-}ctstate\ S\ s\text{-}tag \iff s\text{-}tag \in S$

fun *ctstate-conjunct* :: $ctstate\ set \Rightarrow ctstate\ set \Rightarrow ctstate\ set\ option$ **where**
 $ctstate\text{-}conjunct\ S1\ S2 = (if\ S1 \cap S2 = \{\} \text{ then } None \text{ else } Some\ (S1 \cap S2))$

value[code] *ctstate-conjunct* {*CT-Established*, *CT-New*} {*CT-New*}

lemma *ctstate-conjunct-correct*: $match\text{-}ctstate\ S1\ pkt \wedge match\text{-}ctstate\ S2\ pkt \iff$

$(case\ ctstate\text{-}conjunct\ S1\ S2\ of\ None \Rightarrow False \mid Some\ S' \Rightarrow match\text{-}ctstate\ S'\ pkt)$
<proof>

lemma *UNIV-ctstate*: $UNIV = \{CT\text{-}New, CT\text{-}Established, CT\text{-}Related, CT\text{-}Untracked, CT\text{-}Invalid\}$ *<proof>*

instance *ctstate* :: *finite*

<proof>

lemma *finite* ($S :: ctstate\ set$) $\langle proof \rangle$

instantiation *ctstate* :: *enum*

begin

definition *enum-ctstate* = [*CT-New*, *CT-Established*, *CT-Related*, *CT-Untracked*, *CT-Invalid*]

definition *enum-all-ctstate* $P \longleftrightarrow P\ CT\text{-New} \wedge P\ CT\text{-Established} \wedge P\ CT\text{-Related} \wedge P\ CT\text{-Untracked} \wedge P\ CT\text{-Invalid}$

definition *enum-ex-ctstate* $P \longleftrightarrow P\ CT\text{-New} \vee P\ CT\text{-Established} \vee P\ CT\text{-Related} \vee P\ CT\text{-Untracked} \vee P\ CT\text{-Invalid}$

instance $\langle proof \rangle$

end

definition *ctstate-is-UNIV* :: *ctstate set* \Rightarrow *bool* **where**

ctstate-is-UNIV $c \equiv CT\text{-New} \in c \wedge CT\text{-Established} \in c \wedge CT\text{-Related} \in c \wedge CT\text{-Untracked} \in c \wedge CT\text{-Invalid} \in c$

lemma *ctstate-is-UNIV*: *ctstate-is-UNIV* $c \longleftrightarrow c = UNIV$
 $\langle proof \rangle$

value[*code*] *ctstate-is-UNIV* {*CT-Established*}

fun *ctstate-toString* :: *ctstate* \Rightarrow *string* **where**

ctstate-toString *CT-New* = "NEW" |
ctstate-toString *CT-Established* = "ESTABLISHED" |
ctstate-toString *CT-Related* = "RELATED" |
ctstate-toString *CT-Untracked* = "UNTRACKED" |
ctstate-toString *CT-Invalid* = "INVALID"

definition *ctstate-set-toString* :: *ctstate set* \Rightarrow *string* **where**

ctstate-set-toString $S = list\text{-separated-toString}\ " , " ctstate\text{-toString}\ (enum\text{-set-to-list}\ S)$

lemma *ctstate-set-toString* {*CT-New*, *CT-New*, *CT-Established*} = "NEW,ESTABLISHED"
 $\langle proof \rangle$

end

theory *Tagged-Packet*

imports *Simple-Firewall.Simple-Packet Conntrack-State*

begin

12 Tagged Simple Packet

Packet constants are prefixed with p

A packet tagged with the following phantom fields: conntrack connection state

The idea to tag the connection state into the packet is sound. See `../Semantics_Stateful.thy`

```
record (overloaded) 'i tagged-packet = 'i::len simple-packet +
      p-tag-ctstate :: ctstate
```

```
value (
  p-iiface = "eth1", p-oiface = "",
  p-src = 0, p-dst = 0,
  p-proto = TCP, p-sport = 0, p-dport = 0,
  p-tcp-flags = {TCP-SYN},
  p-payload = "arbitrary payload",
  p-tag-ctstate = CT-New
):: 32 tagged-packet
```

definition *simple-packet-tag*

```
:: ctstate  $\Rightarrow$  ('i::len, 'a) simple-packet-scheme  $\Rightarrow$  ('i::len, 'a) tagged-packet-scheme
```

where

```
simple-packet-tag ct-state p  $\equiv$ 
  (p-iiface = p-iiface p, p-oiface = p-oiface p, p-src = p-src p, p-dst = p-dst p,
  p-proto = p-proto p,
  p-sport = p-sport p, p-dport = p-dport p, p-tcp-flags = p-tcp-flags p,
  p-payload = p-payload p,
  p-tag-ctstate = ct-state,
  ... = simple-packet.more p)
```

definition *tagged-packet-untag*

```
:: ('i::len, 'a) tagged-packet-scheme  $\Rightarrow$  ('i::len, 'a) simple-packet-scheme where
```

```
tagged-packet-untag p  $\equiv$ 
  (p-iiface = p-iiface p, p-oiface = p-oiface p, p-src = p-src p, p-dst = p-dst p,
  p-proto = p-proto p,
  p-sport = p-sport p, p-dport = p-dport p, p-tcp-flags = p-tcp-flags p,
  p-payload = p-payload p,
  ... = tagged-packet.more p)
```

lemma *tagged-packet-untag (simple-packet-tag ct-state p) = p*

```
simple-packet-tag ct-state (tagged-packet-untag p) = p (p-tag-ctstate :=
ct-state)
<proof>
```

end

```

theory Common-Primitive-Syntax
imports ../Datatype-Selectors
         IpAddresses
         Simple-Firewall.Iface
         L4-Protocol-Flags Ports Tagged-Packet Conntrack-State
begin

```

13 Primitive Matchers: Interfaces, IP Space, Layer 4 Ports Matcher

Primitive Match Conditions which only support interfaces, IPv4 addresses, layer 4 protocols, and layer 4 ports.

```

context
  notes [[typedef-overloaded]]
begin
  datatype 'i common-primitive =
    is-Src: Src (src-sel: 'i::len ipt-iprange) |
    is-Dst: Dst (dst-sel: 'i::len ipt-iprange) |
    is-Iface: Iiface (iiface-sel: iface) |
    is-Oiface: Oiface (oiface-sel: iface) |
    is-Prot: Prot (prot-sel: protocol) |
    is-Src-Ports: Src-Ports (src-ports-sel: ipt-l4-ports) |
    is-Dst-Ports: Dst-Ports (dst-ports-sel: ipt-l4-ports) |
    is-MultiportPorts: MultiportPorts (multiportports-sel: ipt-l4-ports) |
    is-L4-Flags: L4-Flags (l4-flags-sel: ipt-tcp-flags) |
    is-CT-State: CT-State (ct-state-sel: ctstate set) |
    is-Extra: Extra (extra-sel: string)
end

```

```

lemma wf-disc-sel-common-primitive:
  wf-disc-sel (is-Src-Ports, src-ports-sel) Src-Ports
  wf-disc-sel (is-Dst-Ports, dst-ports-sel) Dst-Ports
  wf-disc-sel (is-Src, src-sel) Src
  wf-disc-sel (is-Dst, dst-sel) Dst
  wf-disc-sel (is-Iiface, iiface-sel) Iiface
  wf-disc-sel (is-Oiface, oiface-sel) Oiface
  wf-disc-sel (is-Prot, prot-sel) Prot
  wf-disc-sel (is-L4-Flags, l4-flags-sel) L4-Flags
  wf-disc-sel (is-CT-State, ct-state-sel) CT-State
  wf-disc-sel (is-Extra, extra-sel) Extra
  wf-disc-sel (is-MultiportPorts, multiportports-sel) MultiportPorts
  <proof>
  value (|p-iiface = "eth0", p-oiface = "eth1",
         p-src = ipv4addr-of-dotdecimal (192,168,2,45), p-dst = ipv4addr-of-dotdecimal
         (173,194,112,111),
         p-proto = TCP, p-sport = 2065, p-dport = 80, p-tcp-flags = {TCP-ACK},

```

```
p-payload = "GET / HTTP/1.0",
p-tag-ctstate = CT-Established) :: 32 tagged-packet
```

```
end
theory Unknown-Match-Tacs
imports Matching-Ternary
begin
```

14 Approximate Matching Tactics

in-doubt-tactics

```
fun in-doubt-allow :: 'packet unknown-match-tac where
  in-doubt-allow Accept - = True |
  in-doubt-allow Drop - = False |
  in-doubt-allow Reject - = False |
  in-doubt-allow - - = undefined
```

```
lemma wf-in-doubt-allow: wf-unknown-match-tac in-doubt-allow
  <proof>
```

```
fun in-doubt-deny :: 'packet unknown-match-tac where
  in-doubt-deny Accept - = False |
  in-doubt-deny Drop - = True |
  in-doubt-deny Reject - = True |
  in-doubt-deny - - = undefined
```

```
lemma wf-in-doubt-deny: wf-unknown-match-tac in-doubt-deny
  <proof>
```

```
lemma packet-independent-unknown-match-tacs:
  packet-independent- $\alpha$  in-doubt-allow
  packet-independent- $\alpha$  in-doubt-deny
  <proof>
```

```
lemma Drop-neq-Accept-unknown-match-tacs:
  in-doubt-allow Drop  $\neq$  in-doubt-allow Accept
```

in-doubt-deny Drop \neq *in-doubt-deny Accept*
 ⟨proof⟩

corollary *matches-induction-case-MatchNot-in-doubt-allow:*

$\forall a. \text{matches } (\beta, \text{in-doubt-allow}) m' a p = \text{matches } (\beta, \text{in-doubt-allow}) m a p$
 \implies
 $\text{matches } (\beta, \text{in-doubt-allow}) (\text{MatchNot } m') a p = \text{matches } (\beta, \text{in-doubt-allow})$
 $(\text{MatchNot } m) a p$
 ⟨proof⟩

corollary *matches-induction-case-MatchNot-in-doubt-deny:*

$\forall a. \text{matches } (\beta, \text{in-doubt-deny}) m' a p = \text{matches } (\beta, \text{in-doubt-deny}) m a p$
 \implies
 $\text{matches } (\beta, \text{in-doubt-deny}) (\text{MatchNot } m') a p = \text{matches } (\beta, \text{in-doubt-deny})$
 $(\text{MatchNot } m) a p$
 ⟨proof⟩

end

theory *Common-Primitive-Matcher-Generic*

imports *../Semantics-Ternary/Semantics-Ternary*

Common-Primitive-Syntax

../Semantics-Ternary/Unknown-Match-Tacs

begin

14.1 A Generic primitive matcher: Agnostic of IP Addresses

Generalized Definition agnostic of IP Addresses fro IPv4 and IPv6

locale *primitive-matcher-generic* =

fixes $\beta :: ('i::\text{len common-primitive}, ('i::\text{len}, 'a) \text{tagged-packet-scheme}) \text{exact-match-tac}$
assumes *IIface*: $\forall p i. \beta (\text{IIface } i) p = \text{bool-to-ternary } (\text{match-iface } i (p\text{-iiface } p))$

and *OIface*: $\forall p i. \beta (\text{OIface } i) p = \text{bool-to-ternary } (\text{match-iface } i (p\text{-oiface } p))$

and *Prot*: $\forall p \text{proto}. \beta (\text{Prot } \text{proto}) p = \text{bool-to-ternary } (\text{match-proto } \text{proto } (p\text{-proto } p))$

and *Src-Ports*: $\forall p \text{proto } ps. \beta (\text{Src-Ports } (L4Ports \text{proto } ps)) p = \text{bool-to-ternary } (p\text{-proto } p \wedge p\text{-sport } p \in \text{ports-to-set } ps)$

and *Dst-Ports*: $\forall p \text{proto } ps. \beta (\text{Dst-Ports } (L4Ports \text{proto } ps)) p = \text{bool-to-ternary } (p\text{-proto } p \wedge p\text{-dport } p \in \text{ports-to-set } ps)$

— *-m multiport -ports* matches source or destination port

and *MultiportsPorts*: $\forall p \text{proto } ps. \beta (\text{MultiportsPorts } (L4Ports \text{proto } ps)) p = \text{bool-to-ternary } (p\text{-proto } p \wedge (p\text{-sport } p \in \text{ports-to-set } ps \vee p\text{-dport } p \in \text{ports-to-set } ps))$

and *L4-Flags*: $\forall p \text{flags}. \beta (L4-Flags \text{flags}) p = \text{bool-to-ternary } (\text{match-tcp-flags } \text{flags } (p\text{-tcp-flags } p))$

and *CT-State*: $\forall p S. \beta (\text{CT-State } S) p = \text{bool-to-ternary } (\text{match-ctstate } S (p\text{-tag-ctstate } p))$

and *Extra*: $\forall p \text{ str. } \beta (\text{Extra str}) p = \text{TernaryUnknown}$
begin

lemma *Iface-single*:

$\text{matches } (\beta, \alpha) (\text{Match } (\text{Iface } X)) a p \longleftrightarrow \text{match-iface } X (p\text{-iface } p)$
 $\text{matches } (\beta, \alpha) (\text{Match } (\text{Oiface } X)) a p \longleftrightarrow \text{match-iface } X (p\text{-oiface } p)$
 $\langle \text{proof} \rangle$

Since matching on the iface cannot be *TernaryUnknown**, we can pull out negations.

lemma *Iface-single-not*:

$\text{matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{Iface } X))) a p \longleftrightarrow \neg \text{match-iface } X (p\text{-iface } p)$
 $\text{matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{Oiface } X))) a p \longleftrightarrow \neg \text{match-iface } X (p\text{-oiface } p)$
 $\langle \text{proof} \rangle$

lemma *Prot-single*:

$\text{matches } (\beta, \alpha) (\text{Match } (\text{Prot } X)) a p \longleftrightarrow \text{match-proto } X (p\text{-proto } p)$
 $\langle \text{proof} \rangle$

lemma *Prot-single-not*:

$\text{matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{Prot } X))) a p \longleftrightarrow \neg \text{match-proto } X (p\text{-proto } p)$
 $\langle \text{proof} \rangle$

lemma *Ports-single*:

$\text{matches } (\beta, \alpha) (\text{Match } (\text{Src-Ports } (L4Ports \text{ proto } ps))) a p \longleftrightarrow \text{proto} = p\text{-proto}$
 $p \wedge p\text{-sport } p \in \text{ports-to-set } ps$
 $\text{matches } (\beta, \alpha) (\text{Match } (\text{Dst-Ports } (L4Ports \text{ proto } ps))) a p \longleftrightarrow \text{proto} = p\text{-proto}$
 $p \wedge p\text{-dport } p \in \text{ports-to-set } ps$
 $\langle \text{proof} \rangle$

lemma *Ports-single-not*:

$\text{matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{Src-Ports } (L4Ports \text{ proto } ps)))) a p \longleftrightarrow$
 $\text{proto} \neq p\text{-proto } p \vee p\text{-sport } p \notin \text{ports-to-set } ps$
 $\text{matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{Dst-Ports } (L4Ports \text{ proto } ps)))) a p \longleftrightarrow$
 $\text{proto} \neq p\text{-proto } p \vee p\text{-dport } p \notin \text{ports-to-set } ps$
 $\langle \text{proof} \rangle$

Ports are dependent matches. They always match on the protocol too

lemma *Ports-single-rewrite-Prot*:

$\text{matches } (\beta, \alpha) (\text{Match } (\text{Src-Ports } (L4Ports \text{ proto } ps))) a p \longleftrightarrow$
 $\text{matches } (\beta, \alpha) (\text{Match } (\text{Prot } (\text{Proto } \text{proto}))) a p \wedge p\text{-sport } p \in \text{ports-to-set } ps$
 $\text{matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{Src-Ports } (L4Ports \text{ proto } ps)))) a p \longleftrightarrow$
 $\text{matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{Prot } (\text{Proto } \text{proto})))) a p \vee p\text{-sport } p \notin$
 $\text{ports-to-set } ps$
 $\text{matches } (\beta, \alpha) (\text{Match } (\text{Dst-Ports } (L4Ports \text{ proto } ps))) a p \longleftrightarrow$
 $\text{matches } (\beta, \alpha) (\text{Match } (\text{Prot } (\text{Proto } \text{proto}))) a p \wedge p\text{-dport } p \in \text{ports-to-set } ps$
 $\text{matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{Dst-Ports } (L4Ports \text{ proto } ps)))) a p \longleftrightarrow$
 $\text{matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{Prot } (\text{Proto } \text{proto})))) a p \vee p\text{-dport } p \notin$
 $\text{ports-to-set } ps$

<proof>

lemma *multiports-disjunction:*

$(\exists rg \in \text{set } spts. \text{ matches } (\beta, \alpha) (\text{Match } (\text{Src-Ports } (L4Ports \text{ proto } [rg]))) a p)$
 $\longleftrightarrow \text{ matches } (\beta, \alpha) (\text{Match } (\text{Src-Ports } (L4Ports \text{ proto } spts))) a p$
 $(\exists rg \in \text{set } dpts. \text{ matches } (\beta, \alpha) (\text{Match } (\text{Dst-Ports } (L4Ports \text{ proto } [rg]))) a$
 $p) \longleftrightarrow \text{ matches } (\beta, \alpha) (\text{Match } (\text{Dst-Ports } (L4Ports \text{ proto } dpts))) a p$
<proof>

lemma *MultiportPorts-single-rewrite:*

$\text{ matches } (\beta, \alpha) (\text{Match } (\text{MultiportPorts } ports)) a p \longleftrightarrow$
 $\text{ matches } (\beta, \alpha) (\text{Match } (\text{Src-Ports } ports)) a p \vee \text{ matches } (\beta, \alpha) (\text{Match}$
 $(\text{Dst-Ports } ports)) a p$
<proof>

lemma *MultiportPorts-single-rewrite-MatchOr:*

$\text{ matches } (\beta, \alpha) (\text{Match } (\text{MultiportPorts } ports)) a p \longleftrightarrow$
 $\text{ matches } (\beta, \alpha) (\text{MatchOr } (\text{Match } (\text{Src-Ports } ports)) (\text{Match } (\text{Dst-Ports } ports)))$
 $a p$
<proof>

lemma *MultiportPorts-single-not-rewrite-MatchAnd:*

$\text{ matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{MultiportPorts } ports))) a p \longleftrightarrow$
 $\text{ matches } (\beta, \alpha) (\text{MatchAnd } (\text{MatchNot } (\text{Match } (\text{Src-Ports } ports))) (\text{MatchNot}$
 $(\text{Match } (\text{Dst-Ports } ports)))) a p$
<proof>

lemma *MultiportPorts-single-not-rewrite:*

$\text{ matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{MultiportPorts } ports))) a p \longleftrightarrow$
 $\neg \text{ matches } (\beta, \alpha) (\text{Match } (\text{Src-Ports } ports)) a p \wedge \neg \text{ matches } (\beta, \alpha) (\text{Match}$
 $(\text{Dst-Ports } ports)) a p$
<proof>

lemma *Extra-single:*

$\text{ matches } (\beta, \alpha) (\text{Match } (\text{Extra } str)) a p \longleftrightarrow \alpha a p$
<proof>

lemma *Extra-single-not:* — ternary logic, $\neg \text{ unknown} = \text{ unknown}$

$\text{ matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{Extra } str))) a p \longleftrightarrow \alpha a p$
<proof>

end

14.2 Basic optimisations

Compress many *Extra* expressions to one expression.

fun *compress-extra* :: 'i::len common-primitive match-expr \Rightarrow 'i common-primitive
match-expr **where**

compress-extra (Match x) = Match x |
compress-extra (MatchNot (Match (Extra e))) = Match (Extra ("NOT ("@e@"))

|

```

compress-extra (MatchNot m) = (MatchNot (compress-extra m)) |

compress-extra (MatchAnd (Match (Extra e1)) m2) = (case compress-extra m2
of Match (Extra e2) ⇒ Match (Extra (e1@'' '@e2)) | MatchAny ⇒ Match (Extra
e1) | m2' ⇒ MatchAnd (Match (Extra e1)) m2') |
compress-extra (MatchAnd m1 m2) = MatchAnd (compress-extra m1) (compress-extra
m2) |

compress-extra MatchAny = MatchAny

thm compress-extra.simps

value [nbe] compress-extra (MatchAnd (Match (Extra "foo")) (Match (Extra
"bar")))
value [nbe] compress-extra (MatchAnd (Match (Extra "foo")) (MatchNot (Match
(Extra "bar"))))
value [nbe] compress-extra (MatchAnd (Match (Extra "--m")) (MatchAnd (Match
(Extra "addrtype")) (MatchAnd (Match (Extra "--dst-type")) (MatchAnd (Match
(Extra "BROADCAST")) MatchAny))))

lemma compress-extra-correct-matchexpr:
fixes β::('i::len common-primitive, ('i::len, 'a) tagged-packet-scheme) exact-match-tac
assumes generic: primitive-matcher-generic β
shows matches (β, α) m = matches (β, α) (compress-extra m)
<proof>

end
theory Common-Primitive-Matcher
imports Common-Primitive-Matcher-Generic
begin

```

14.3 Primitive Matchers: IP Port Iface Matcher

```

fun common-matcher :: ('i::len common-primitive, ('i, 'a) tagged-packet-scheme)
exact-match-tac where
common-matcher (Iiface i) p = bool-to-ternary (match-iface i (p-iiface p)) |
common-matcher (Oiface i) p = bool-to-ternary (match-iface i (p-oiface p)) |

common-matcher (Src ip) p = bool-to-ternary (p-src p ∈ ipt-iprange-to-set ip) |
common-matcher (Dst ip) p = bool-to-ternary (p-dst p ∈ ipt-iprange-to-set ip) |

common-matcher (Prot proto) p = bool-to-ternary (match-proto proto (p-proto
p)) |

common-matcher (Src-Ports (L4Ports proto ps)) p = bool-to-ternary (proto =
p-proto p ∧ p-sport p ∈ ports-to-set ps) |
common-matcher (Dst-Ports (L4Ports proto ps)) p = bool-to-ternary (proto =
p-proto p ∧ p-dport p ∈ ports-to-set ps) |

```

*common-matcher (MultiportPorts (L4Ports proto ps)) p = bool-to-ternary (proto = p-*proto* p ∧ (p-*sport* p ∈ ports-to-set ps ∨ p-*dport* p ∈ ports-to-set ps)) |*

*common-matcher (L4-Flags flags) p = bool-to-ternary (match-tcp-flags flags (p-*tcp-flags* p)) |*

*common-matcher (CT-State S) p = bool-to-ternary (match-ctstate S (p-*tag-ctstate* p)) |*

common-matcher (Extra -) p = TernaryUnknown

lemma *packet-independent-β-unknown-common-matcher: packet-independent-β-unknown common-matcher*
 ⟨*proof*⟩

lemma *primitive-matcher-generic-common-matcher: primitive-matcher-generic common-matcher*
 ⟨*proof*⟩

Warning: beware of the sloppy term ‘empty’ portrange

An ‘empty’ port range means it can never match! Basically, *MatchNot (Match (Src-Ports (L4Ports proto [(0, 65535)])))* is False

lemma \neg *matches (common-matcher, α) (MatchNot (Match (Src-Ports (L4Ports TCP [(0,65535)]))) a*
 (p-*iiface* = "eth0", p-*oiface* = "eth1",
 p-*src* = ipv4addr-of-dotdecimal (192,168,2,45), p-*dst* = ipv4addr-of-dotdecimal (173,194,112,111),
 p-*proto* = TCP, p-*sport* = 2065, p-*dport* = 80, p-*tcp-flags* = {},
 p-*payload* = "", p-*tag-ctstate* = CT-New)
 ⟨*proof*⟩

An ‘empty’ port range means it always matches! Basically, *MatchNot (Match (Src-Ports (L4Ports any [])))* is True. This corresponds to firewall behavior, but usually you cannot specify an empty portrange in firewalls, but omission of portrange means no-port-restrictions, i.e. every port matches.

lemma *matches (common-matcher, α) (MatchNot (Match (Src-Ports (L4Ports any []))) a*
 (p-*iiface* = "eth0", p-*oiface* = "eth1",
 p-*src* = ipv4addr-of-dotdecimal (192,168,2,45), p-*dst* = ipv4addr-of-dotdecimal (173,194,112,111),
 p-*proto* = TCP, p-*sport* = 2065, p-*dport* = 80, p-*tcp-flags* = {},
 p-*payload* = "", p-*tag-ctstate* = CT-New)
 ⟨*proof*⟩

If not a corner case, portrange matching is straight forward.

lemma *matches* (*common-matcher*, α) (*Match* (*Src-Ports* (*L4Ports* *TCP* [(1024,4096), (9999, 65535)]))) *a*
 (*p-iiface* = "eth0", *p-oiface* = "eth1",
p-src = *ipv4addr-of-dotdecimal* (192,168,2,45), *p-dst* = *ipv4addr-of-dotdecimal*
 (173,194,112,111),
p-proto = *TCP*, *p-sport* = 2065, *p-dport* = 80, *p-tcp-flags* = {},
p-payload = "", *p-tag-ctstate* = *CT-New*)
 \neg *matches* (*common-matcher*, α) (*Match* (*Src-Ports* (*L4Ports* *TCP* [(1024,4096), (9999, 65535)]))) *a*
 (*p-iiface* = "eth0", *p-oiface* = "eth1",
p-src = *ipv4addr-of-dotdecimal* (192,168,2,45), *p-dst* = *ipv4addr-of-dotdecimal*
 (173,194,112,111),
p-proto = *TCP*, *p-sport* = 5000, *p-dport* = 80, *p-tcp-flags* = {},
p-payload = "", *p-tag-ctstate* = *CT-New*)
 \neg *matches* (*common-matcher*, α) (*MatchNot* (*Match* (*Src-Ports* (*L4Ports*
TCP [(1024,4096), (9999, 65535)])))) *a*
 (*p-iiface* = "eth0", *p-oiface* = "eth1",
p-src = *ipv4addr-of-dotdecimal* (192,168,2,45), *p-dst* = *ipv4addr-of-dotdecimal*
 (173,194,112,111),
p-proto = *TCP*, *p-sport* = 2065, *p-dport* = 80, *p-tcp-flags* = {},
p-payload = "", *p-tag-ctstate* = *CT-New*)
 ⟨*proof*⟩

Lemmas when matching on *Src* or *Dst*

lemma *common-matcher-SrcDst-defined*:
common-matcher (*Src* *m*) $p \neq$ *TernaryUnknown*
common-matcher (*Dst* *m*) $p \neq$ *TernaryUnknown*
common-matcher (*Src-Ports* *ps*) $p \neq$ *TernaryUnknown*
common-matcher (*Dst-Ports* *ps*) $p \neq$ *TernaryUnknown*
common-matcher (*MultiportPorts* *ps*) $p \neq$ *TernaryUnknown*
 ⟨*proof*⟩

lemma *common-matcher-SrcDst-defined-simp*:
common-matcher (*Src* *x*) $p \neq$ *TernaryFalse* \longleftrightarrow *common-matcher* (*Src* *x*) $p =$
TernaryTrue
common-matcher (*Dst* *x*) $p \neq$ *TernaryFalse* \longleftrightarrow *common-matcher* (*Dst* *x*) $p =$
TernaryTrue
 ⟨*proof*⟩

lemma *match-simplematcher-SrcDst*:
matches (*common-matcher*, α) (*Match* (*Src* *X*)) $a p \longleftrightarrow p\text{-src } p \in \textit{ipt-irange-to-set}$
X
matches (*common-matcher*, α) (*Match* (*Dst* *X*)) $a p \longleftrightarrow p\text{-dst } p \in \textit{ipt-irange-to-set}$
X
 ⟨*proof*⟩

lemma *match-simplematcher-SrcDst-not*:
matches (*common-matcher*, α) (*MatchNot* (*Match* (*Src* *X*))) $a p \longleftrightarrow p\text{-src } p \notin$
ipt-irange-to-set X
matches (*common-matcher*, α) (*MatchNot* (*Match* (*Dst* *X*))) $a p \longleftrightarrow p\text{-dst } p \notin$
ipt-irange-to-set X

ipt-iprange-to-set X
 ⟨proof⟩

lemma *common-matcher-SrcDst-Inter:*

$(\forall m \in \text{set } X. \text{matches } (\text{common-matcher}, \alpha) (\text{Match } (\text{Src } m)) a p) \longleftrightarrow p\text{-src } p \in$
 $(\bigcap x \in \text{set } X. \text{ipt-iprange-to-set } x)$
 $(\forall m \in \text{set } X. \text{matches } (\text{common-matcher}, \alpha) (\text{Match } (\text{Dst } m)) a p) \longleftrightarrow p\text{-dst } p \in$
 $(\bigcap x \in \text{set } X. \text{ipt-iprange-to-set } x)$
 ⟨proof⟩

14.4 Basic optimisations

Perform very basic optimization. Remove matches to primitives which are essentially *MatchAny*

fun *optimize-primitive-univ* :: 'i::len common-primitive match-expr ⇒ 'i common-primitive match-expr **where**

optimize-primitive-univ (Match (Src (IpAddrNetmask - 0))) = MatchAny |
optimize-primitive-univ (Match (Dst (IpAddrNetmask - 0))) = MatchAny |

optimize-primitive-univ (Match (Iiface iface)) = (if iface = ifaceAny then MatchAny else (Match (Iiface iface))) |

optimize-primitive-univ (Match (Oiface iface)) = (if iface = ifaceAny then MatchAny else (Match (Oiface iface))) |

optimize-primitive-univ (Match (Prot ProtoAny)) = MatchAny |

optimize-primitive-univ (Match (L4-Flags (TCP-Flags m c))) = (if m = {} ∧ c = {} then MatchAny else (Match (L4-Flags (TCP-Flags m c)))) |

optimize-primitive-univ (Match (CT-State ctstate)) = (if ctstate-is-UNIV ctstate then MatchAny else Match (CT-State ctstate)) |

optimize-primitive-univ (Match m) = Match m |

optimize-primitive-univ (MatchNot m) = (MatchNot (optimize-primitive-univ m)) |

optimize-primitive-univ (MatchAnd m1 m2) = MatchAnd (optimize-primitive-univ m1) (optimize-primitive-univ m2) |

optimize-primitive-univ MatchAny = MatchAny

lemma *optimize-primitive-univ-unchanged-primitives:*

optimize-primitive-univ (Match a) = (Match a) ∨ *optimize-primitive-univ* (Match a) = MatchAny
 ⟨proof⟩

lemma *optimize-primitive-univ-correct-matchexpr:* **fixes** m::'i::len common-primitive match-expr

shows matches (common-matcher, α) m = matches (common-matcher, α) (optimize-primitive-univ m)
 ⟨proof⟩

corollary *optimize-primitive-univ-correct: approximating-bigstep-fun (common-matcher, α) p (optimize-matches optimize-primitive-univ rs) s = approximating-bigstep-fun (common-matcher, α) p rs s*
 ⟨proof⟩

14.5 Abstracting over unknowns

remove *Extra* (i.e. *TernaryUnknown*) match expressions

fun *upper-closure-matchexpr* :: *action* \Rightarrow 'i::len *common-primitive match-expr* \Rightarrow 'i *common-primitive match-expr* **where**
upper-closure-matchexpr - *MatchAny* = *MatchAny* |
upper-closure-matchexpr *Accept* (*Match* (*Extra* -)) = *MatchAny* |
upper-closure-matchexpr *Reject* (*Match* (*Extra* -)) = *MatchNot MatchAny* |
upper-closure-matchexpr *Drop* (*Match* (*Extra* -)) = *MatchNot MatchAny* |
upper-closure-matchexpr - (*Match* *m*) = *Match* *m* |
upper-closure-matchexpr *Accept* (*MatchNot* (*Match* (*Extra* -))) = *MatchAny* |
upper-closure-matchexpr *Drop* (*MatchNot* (*Match* (*Extra* -))) = *MatchNot MatchAny* |
upper-closure-matchexpr *Reject* (*MatchNot* (*Match* (*Extra* -))) = *MatchNot MatchAny* |
upper-closure-matchexpr *a* (*MatchNot* (*MatchNot* *m*)) = *upper-closure-matchexpr* *a* *m* |
upper-closure-matchexpr *a* (*MatchNot* (*MatchAnd* *m1* *m2*)) =
 (*let* *m1'* = *upper-closure-matchexpr* *a* (*MatchNot* *m1*); *m2'* = *upper-closure-matchexpr* *a* (*MatchNot* *m2*) *in*
 (*if* *m1'* = *MatchAny* \vee *m2'* = *MatchAny*
then *MatchAny*
else
 if *m1'* = *MatchNot MatchAny* *then* *m2'* *else*
 if *m2'* = *MatchNot MatchAny* *then* *m1'*
else
 MatchNot (*MatchAnd* (*MatchNot* *m1'*) (*MatchNot* *m2'*)))
) |
upper-closure-matchexpr - (*MatchNot* *m*) = *MatchNot* *m* |
upper-closure-matchexpr *a* (*MatchAnd* *m1* *m2*) = *MatchAnd* (*upper-closure-matchexpr* *a* *m1*) (*upper-closure-matchexpr* *a* *m2*)

lemma *upper-closure-matchexpr-generic*:
a = *Accept* \vee *a* = *Drop* \implies *remove-unknowns-generic* (*common-matcher*, *in-doubt-allow*) *a* *m* = *upper-closure-matchexpr* *a* *m*
 ⟨proof⟩

fun *lower-closure-matchexpr* :: *action* \Rightarrow 'i::len *common-primitive match-expr* \Rightarrow 'i *common-primitive match-expr* **where**
lower-closure-matchexpr - *MatchAny* = *MatchAny* |
lower-closure-matchexpr *Accept* (*Match* (*Extra* -)) = *MatchNot MatchAny* |
lower-closure-matchexpr *Reject* (*Match* (*Extra* -)) = *MatchAny* |
lower-closure-matchexpr *Drop* (*Match* (*Extra* -)) = *MatchAny* |

```

lower-closure-matchexpr - (Match m) = Match m |
lower-closure-matchexpr Accept (MatchNot (Match (Extra -))) = MatchNot
MatchAny |
lower-closure-matchexpr Drop (MatchNot (Match (Extra -))) = MatchAny |
lower-closure-matchexpr Reject (MatchNot (Match (Extra -))) = MatchAny |
lower-closure-matchexpr a (MatchNot (MatchNot m)) = lower-closure-matchexpr
a m |
lower-closure-matchexpr a (MatchNot (MatchAnd m1 m2)) =
(let m1' = lower-closure-matchexpr a (MatchNot m1); m2' = lower-closure-matchexpr
a (MatchNot m2) in
(if m1' = MatchAny ∨ m2' = MatchAny
then MatchAny
else
if m1' = MatchNot MatchAny then m2' else
if m2' = MatchNot MatchAny then m1'
else
MatchNot (MatchAnd (MatchNot m1') (MatchNot m2'))
) |
lower-closure-matchexpr - (MatchNot m) = MatchNot m |
lower-closure-matchexpr a (MatchAnd m1 m2) = MatchAnd (lower-closure-matchexpr
a m1) (lower-closure-matchexpr a m2)

```

lemma *lower-closure-matchexpr-generic*:

$a = \text{Accept} \vee a = \text{Drop} \implies \text{remove-unknowns-generic } (\text{common-matcher},$
in-doubt-deny) $a\ m = \text{lower-closure-matchexpr } a\ m$
 $\langle \text{proof} \rangle$

end

theory *Example-Semantics*

imports *Call-Return-Unfolding Primitive-Matchers/Common-Primitive-Matcher*

begin

15 Examples Big Step Semantics

We use a primitive matcher which always applies. We don't care about matching in this example.

fun *applies-Yes* :: ('a, 'p) *matcher* **where**

applies-Yes $m\ p = \text{True}$

lemma_[simp]: *Semantics.matches* *applies-Yes* *MatchAny* $p\ \langle \text{proof} \rangle$

lemma_[simp]: *Semantics.matches* *applies-Yes* (*Match* e) $p\ \langle \text{proof} \rangle$

definition $m = \text{Match } (\text{Src } (\text{IpAddr } (0::\text{ipv4addr})))$

lemma_[simp]: *Semantics.matches* *applies-Yes* $m\ p\ \langle \text{proof} \rangle$

lemma ["FORWARD" \mapsto [(*Rule* $m\ \text{Log}$), (*Rule* $m\ \text{Accept}$), (*Rule* $m\ \text{Drop}$)], *applies-Yes*, p +
 ([*Rule* *MatchAny* (*Call* "FORWARD"), *Undecided*] \Rightarrow (*Decision* *FinalAllow*)

<proof>

lemma ["FORWARD" \mapsto [(Rule m Log), (Rule m (Call "foo")), (Rule m Accept)],
"foo" \mapsto [(Rule m Log), (Rule m Return)]],applies-Yes,p \vdash
 \langle [Rule MatchAny (Call "FORWARD")], Undecided $\rangle \Rightarrow$ (Decision FinalAllow)
<proof>

lemma ["FORWARD" \mapsto [Rule m (Call "foo"), Rule m Drop], "foo" \mapsto []],applies-Yes,p \vdash
FinalDeny)
<proof>

lemma ((λ rs. process-call ["FORWARD" \mapsto [Rule m (Call "foo"), Rule m Drop],
"foo" \mapsto []] rs) \sim^2)
[Rule MatchAny (Call "FORWARD")]
= [Rule (MatchAnd MatchAny m) Drop] *<proof>*

hide-const m

definition pkt=(λ p. (p-iface="+", p-oiface="+", p-src=0, p-dst=0,
p-proto=TCP, p-sport=0, p-dport=0, p-tcp-flags = {TCP-SYN},
p-payload="",p-tag-ctstate= CT-New)

We tune the primitive matcher to support everything we need in the example. Note that the undefined cases cannot be handled with these exact semantics!

fun applies-exampleMatchExact :: (32 common-primitive, 32 tagged-packet) matcher
where

applies-exampleMatchExact (Src (IpAddr addr)) p \longleftrightarrow p-src p = addr |
applies-exampleMatchExact (Dst (IpAddr addr)) p \longleftrightarrow p-dst p = addr |
applies-exampleMatchExact (Prot ProtoAny) p \longleftrightarrow True |
applies-exampleMatchExact (Prot (Proto pr)) p \longleftrightarrow p-proto p = pr

lemma ["FORWARD" \mapsto [Rule (MatchAnd (Match (Src (IpAddr 0))) (Match (Dst (IpAddr 0)))) Reject,
Rule (Match (Dst (IpAddr 0))) Log,
Rule (Match (Prot (Proto TCP))) Accept,
Rule (Match (Prot (Proto TCP))) Drop]
],applies-exampleMatchExact, pkt(λ p-src:=(ip_{v4}addr-of-dotdecimal (1,2,3,4)),
p-dst:=(ip_{v4}addr-of-dotdecimal (0,0,0,0)))] \vdash
 \langle [Rule MatchAny (Call "FORWARD")], Undecided $\rangle \Rightarrow$ (Decision FinalAllow)
<proof>

end

theory Alternative-Semantics

imports Semantics

begin

context begin

private inductive *iptables-bigstep-ns* :: 'a ruleset \Rightarrow ('a, 'p) matcher \Rightarrow 'p \Rightarrow 'a rule list \Rightarrow state \Rightarrow state \Rightarrow bool

($\langle -, -, + \langle -, - \rangle \Rightarrow_s \rightarrow$ [60,60,60,20,98,98] 89)

for Γ **and** γ **and** p **where**

skip: $\Gamma, \gamma, p \vdash \langle [], t \rangle \Rightarrow_s t$ |

accept: matches γ m $p \Rightarrow \Gamma, \gamma, p \vdash \langle \text{Rule } m \text{ Accept } \# rs, \text{Undecided} \rangle \Rightarrow_s \text{Decision FinalAllow}$ |

drop: matches γ m $p \Rightarrow \Gamma, \gamma, p \vdash \langle \text{Rule } m \text{ Drop } \# rs, \text{Undecided} \rangle \Rightarrow_s \text{Decision FinalDeny}$ |

reject: matches γ m $p \Rightarrow \Gamma, \gamma, p \vdash \langle \text{Rule } m \text{ Reject } \# rs, \text{Undecided} \rangle \Rightarrow_s \text{Decision FinalDeny}$ |

log: matches γ m $p \Rightarrow \Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_s t \Rightarrow \Gamma, \gamma, p \vdash \langle \text{Rule } m \text{ Log } \# rs, \text{Undecided} \rangle \Rightarrow_s t$ |

empty: matches γ m $p \Rightarrow \Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_s t \Rightarrow \Gamma, \gamma, p \vdash \langle \text{Rule } m \text{ Empty } \# rs, \text{Undecided} \rangle \Rightarrow_s t$ |

nms: \neg matches γ m $p \Rightarrow \Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_s t \Rightarrow \Gamma, \gamma, p \vdash \langle \text{Rule } m \text{ a } \# rs, \text{Undecided} \rangle \Rightarrow_s t$ |

call-return: $\llbracket \text{matches } \gamma \text{ } m \text{ } p; \Gamma \text{ chain} = \text{Some } (rs_1 \text{ @ Rule } m' \text{ Return } \# rs_2); \text{matches } \gamma \text{ } m' \text{ } p; \Gamma, \gamma, p \vdash \langle rs_1, \text{Undecided} \rangle \Rightarrow_s \text{Undecided}; \Gamma, \gamma, p \vdash \langle rrs, \text{Undecided} \rangle \Rightarrow_s t \rrbracket \Rightarrow$
 $\Gamma, \gamma, p \vdash \langle \text{Rule } m \text{ (Call chain) } \# rrs, \text{Undecided} \rangle \Rightarrow_s t$ |

call-result: $\llbracket \text{matches } \gamma \text{ } m \text{ } p; \Gamma \text{ chain} = \text{Some } rs; \Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_s \text{Decision } X \rrbracket \Rightarrow$
 $\Gamma, \gamma, p \vdash \langle \text{Rule } m \text{ (Call chain) } \# rrs, \text{Undecided} \rangle \Rightarrow_s \text{Decision } X$ |

call-no-result: $\llbracket \text{matches } \gamma \text{ } m \text{ } p; \Gamma \text{ chain} = \text{Some } rs; \Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_s \text{Undecided};$

$\Gamma, \gamma, p \vdash \langle rrs, \text{Undecided} \rangle \Rightarrow_s t \rrbracket \Rightarrow$
 $\Gamma, \gamma, p \vdash \langle \text{Rule } m \text{ (Call chain) } \# rrs, \text{Undecided} \rangle \Rightarrow_s t$

private lemma *a*: $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow_s t \Rightarrow \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$

(proof) **lemma** *empty-rs-stateD*: **assumes** $\Gamma, \gamma, p \vdash \langle [], s \rangle \Rightarrow_s t$ **shows** $t = s$

(proof) **lemma** *decided*: $\llbracket \Gamma, \gamma, p \vdash \langle rs_1, \text{Undecided} \rangle \Rightarrow_s \text{Decision } X \rrbracket \Rightarrow \Gamma, \gamma, p \vdash \langle rs_1 \text{ @ } rs_2, \text{Undecided} \rangle \Rightarrow_s \text{Decision } X$

(proof) **lemma** *decided-determ*: $\llbracket \Gamma, \gamma, p \vdash \langle rs_1, s \rangle \Rightarrow_s t; s = \text{Decision } X \rrbracket \Rightarrow t = \text{Decision } X$

(proof) **lemma** *seq-ns*:

$\llbracket \Gamma, \gamma, p \vdash \langle rs_1, \text{Undecided} \rangle \Rightarrow_s t; \Gamma, \gamma, p \vdash \langle rs_2, t \rangle \Rightarrow_s t' \rrbracket \Rightarrow \Gamma, \gamma, p \vdash \langle rs_1 \text{ @ } rs_2, \text{Undecided} \rangle \Rightarrow_s t'$

(proof) **lemma** *b*: $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t \Rightarrow s = \text{Undecided} \Rightarrow \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow_s t$

(proof) **inductive** *iptables-bigstep-nz* :: 'a ruleset \Rightarrow ('a, 'p) matcher \Rightarrow 'p \Rightarrow 'a rule list \Rightarrow state \Rightarrow bool

($\langle -, -, + - \Rightarrow_z \rightarrow$ [60,60,60,20,98] 89)

for Γ **and** γ **and** p **where**

skip: $\Gamma, \gamma, p \vdash [] \Rightarrow_z \text{Undecided} \mid$
accept: $\text{matches } \gamma \ m \ p \Rightarrow \Gamma, \gamma, p \vdash \text{Rule } m \ \text{Accept} \ \# \ rs \Rightarrow_z \text{Decision } \text{FinalAllow} \mid$
drop: $\text{matches } \gamma \ m \ p \Rightarrow \Gamma, \gamma, p \vdash \text{Rule } m \ \text{Drop} \ \# \ rs \Rightarrow_z \text{Decision } \text{FinalDeny} \mid$
reject: $\text{matches } \gamma \ m \ p \Rightarrow \Gamma, \gamma, p \vdash \text{Rule } m \ \text{Reject} \ \# \ rs \Rightarrow_z \text{Decision } \text{FinalDeny} \mid$
log: $\text{matches } \gamma \ m \ p \Rightarrow \Gamma, \gamma, p \vdash rs \Rightarrow_z t \Rightarrow \Gamma, \gamma, p \vdash \text{Rule } m \ \text{Log} \ \# \ rs \Rightarrow_z t \mid$
empty: $\text{matches } \gamma \ m \ p \Rightarrow \Gamma, \gamma, p \vdash rs \Rightarrow_z t \Rightarrow \Gamma, \gamma, p \vdash \text{Rule } m \ \text{Empty} \ \# \ rs \Rightarrow_z t \mid$
nms: $\neg \text{matches } \gamma \ m \ p \Rightarrow \Gamma, \gamma, p \vdash rs \Rightarrow_z t \Rightarrow \Gamma, \gamma, p \vdash \text{Rule } m \ a \ \# \ rs \Rightarrow_z t \mid$
call-return: $\llbracket \text{matches } \gamma \ m \ p; \Gamma \ \text{chain} = \text{Some } (rs_1 \ @ \ \text{Rule } m' \ \text{Return} \ \# \ rs_2);$
 $\text{matches } \gamma \ m' \ p; \Gamma, \gamma, p \vdash rs_1 \Rightarrow_z \text{Undecided}; \Gamma, \gamma, p \vdash rrs \Rightarrow_z t \rrbracket \Rightarrow$
 $\Gamma, \gamma, p \vdash \text{Rule } m \ (\text{Call chain}) \ \# \ rrs \Rightarrow_z t \mid$
call-result: $\llbracket \text{matches } \gamma \ m \ p; \Gamma \ \text{chain} = \text{Some } rs; \Gamma, \gamma, p \vdash rs \Rightarrow_z \text{Decision } X \rrbracket \Rightarrow$
 $\Gamma, \gamma, p \vdash \text{Rule } m \ (\text{Call chain}) \ \# \ rrs \Rightarrow_z \text{Decision } X \mid$
call-no-result: $\llbracket \text{matches } \gamma \ m \ p; \Gamma \ \text{chain} = \text{Some } rs; \Gamma, \gamma, p \vdash rs \Rightarrow_z \text{Undecided};$
 $\Gamma, \gamma, p \vdash rrs \Rightarrow_z t \rrbracket \Rightarrow$
 $\Gamma, \gamma, p \vdash \text{Rule } m \ (\text{Call chain}) \ \# \ rrs \Rightarrow_z t$

private lemma *c*: $\Gamma, \gamma, p \vdash rs \Rightarrow_z t \Rightarrow \Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_s t$
 $\langle \text{proof} \rangle$ **lemma** *d*: $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow_s t \Rightarrow s = \text{Undecided} \Rightarrow \Gamma, \gamma, p \vdash rs \Rightarrow_z t$
 $\langle \text{proof} \rangle$

inductive *iptables-bigstep-r* :: $'a \ \text{ruleset} \Rightarrow ('a, 'p) \ \text{matcher} \Rightarrow 'p \Rightarrow 'a \ \text{rule list}$
 $\Rightarrow \text{state} \Rightarrow \text{bool}$
 $(\langle -, -, + - \Rightarrow_r - \rangle [60, 60, 60, 20, 98] \ 89)$

for Γ and γ and p where

skip: $\Gamma, \gamma, p \vdash [] \Rightarrow_r \text{Undecided} \mid$
accept: $\text{matches } \gamma \ m \ p \Rightarrow \Gamma, \gamma, p \vdash \text{Rule } m \ \text{Accept} \ \# \ rs \Rightarrow_r \text{Decision } \text{FinalAllow} \mid$
drop: $\text{matches } \gamma \ m \ p \Rightarrow \Gamma, \gamma, p \vdash \text{Rule } m \ \text{Drop} \ \# \ rs \Rightarrow_r \text{Decision } \text{FinalDeny} \mid$
reject: $\text{matches } \gamma \ m \ p \Rightarrow \Gamma, \gamma, p \vdash \text{Rule } m \ \text{Reject} \ \# \ rs \Rightarrow_r \text{Decision } \text{FinalDeny} \mid$
return: $\text{matches } \gamma \ m \ p \Rightarrow \Gamma, \gamma, p \vdash \text{Rule } m \ \text{Return} \ \# \ rs \Rightarrow_r \text{Undecided} \mid$
log: $\Gamma, \gamma, p \vdash rs \Rightarrow_r t \Rightarrow \Gamma, \gamma, p \vdash \text{Rule } m \ \text{Log} \ \# \ rs \Rightarrow_r t \mid$
empty: $\Gamma, \gamma, p \vdash rs \Rightarrow_r t \Rightarrow \Gamma, \gamma, p \vdash \text{Rule } m \ \text{Empty} \ \# \ rs \Rightarrow_r t \mid$
nms: $\neg \text{matches } \gamma \ m \ p \Rightarrow \Gamma, \gamma, p \vdash rs \Rightarrow_r t \Rightarrow \Gamma, \gamma, p \vdash \text{Rule } m \ a \ \# \ rs \Rightarrow_r t \mid$
call-result: $\llbracket \text{matches } \gamma \ m \ p; \Gamma \ \text{chain} = \text{Some } rs; \Gamma, \gamma, p \vdash rs \Rightarrow_r \text{Decision } X \rrbracket \Rightarrow$
 $\Gamma, \gamma, p \vdash \text{Rule } m \ (\text{Call chain}) \ \# \ rrs \Rightarrow_r \text{Decision } X \mid$
call-no-result: $\llbracket \Gamma \ \text{chain} = \text{Some } rs; \Gamma, \gamma, p \vdash rs \Rightarrow_r \text{Undecided};$
 $\Gamma, \gamma, p \vdash rrs \Rightarrow_r t \rrbracket \Rightarrow$
 $\Gamma, \gamma, p \vdash \text{Rule } m \ (\text{Call chain}) \ \# \ rrs \Rightarrow_r t$

private lemma *returning*: $\llbracket \Gamma, \gamma, p \vdash rs_1 \Rightarrow_r \text{Undecided}; \text{matches } \gamma \ m' \ p \rrbracket$
 $\Rightarrow \Gamma, \gamma, p \vdash rs_1 \ @ \ \text{Rule } m' \ \text{Return} \ \# \ rs_2 \Rightarrow_r \text{Undecided}$
 $\langle \text{proof} \rangle$ **lemma** *e*: $\Gamma, \gamma, p \vdash rs \Rightarrow_z t \Rightarrow s = \text{Undecided} \Rightarrow \Gamma, \gamma, p \vdash rs \Rightarrow_r t$
 $\langle \text{proof} \rangle$

definition *no-call-to c* $rs \equiv (\forall r \in \text{set } rs. \text{case } \text{get-action } r \ \text{of } \text{Call } c' \Rightarrow c \neq c' \mid$
 $- \Rightarrow \text{True})$

definition *all-chains p* $\Gamma \ rs \equiv (p \ rs \ \wedge \ (\forall l \ rs. \Gamma \ l = \text{Some } rs \longrightarrow p \ rs))$

private lemma *all-chains-no-call-upd*: $\text{all-chains } (\text{no-call-to } c) \ \Gamma \ rs \Rightarrow (\Gamma(c \mapsto$

$x)), \gamma, p \vdash rs \Rightarrow_z t \iff \Gamma, \gamma, p \vdash rs \Rightarrow_z t$
 ⟨proof⟩

lemma *updated-call*: $\Gamma(c \mapsto rs), \gamma, p \vdash rs \Rightarrow_z t \implies \text{matches } \gamma \ m \ p \implies \Gamma(c \mapsto rs), \gamma, p \vdash [\text{Rule } m \ (\text{Call } c)] \Rightarrow_z t$

⟨proof⟩ **lemma** *shows*

log-nz: $\Gamma, \gamma, p \vdash rs \Rightarrow_z t \implies \Gamma, \gamma, p \vdash \text{Rule } m \ \text{Log } \# \ rs \Rightarrow_z t$

and *empty-nz*: $\Gamma, \gamma, p \vdash rs \Rightarrow_z t \implies \Gamma, \gamma, p \vdash \text{Rule } m \ \text{Empty } \# \ rs \Rightarrow_z t$

⟨proof⟩ **lemma** *nz-empty-rs-stateD*: **assumes** $\Gamma, \gamma, p \vdash [] \Rightarrow_z t$ **shows** $t = \text{Undecided}$

⟨proof⟩ **lemma** *upd-callD*: $\Gamma(c \mapsto rs), \gamma, p \vdash [\text{Rule } m \ (\text{Call } c)] \Rightarrow_z t \implies \text{matches } \gamma \ m \ p$

$\implies (\Gamma(c \mapsto rs), \gamma, p \vdash rs \Rightarrow_z t \vee (\exists rs_1 \ rs_2 \ m'. rs = rs_1 \ @ \ \text{Rule } m' \ \text{Return } \# \ rs_2 \wedge \text{matches } \gamma \ m' \ p \wedge \Gamma(c \mapsto rs), \gamma, p \vdash rs_1 \Rightarrow_z \text{Undecided} \wedge t = \text{Undecided}))$

⟨proof⟩ **lemma** *partial-fun-upd*: $(f(x \mapsto y)) \ x = \text{Some } y$ ⟨proof⟩

lemma *f*: $\Gamma, \gamma, p \vdash rs \Rightarrow_r t \implies \text{matches } \gamma \ m \ p \implies \text{all-chains (no-call-to } c) \ \Gamma \ rs$
 \implies

$(\Gamma(c \mapsto rs), \gamma, p \vdash [\text{Rule } m \ (\text{Call } c)] \Rightarrow_z t$
 ⟨proof⟩

lemma *r-skip-inv*: $\Gamma, \gamma, p \vdash [] \Rightarrow_r t \implies t = \text{Undecided}$
 ⟨proof⟩

lemma *r-call-eq*: $\Gamma \ c = \text{Some } rs \implies \text{matches } \gamma \ m \ p \implies \Gamma, \gamma, p \vdash [\text{Rule } m \ (\text{Call } c)] \Rightarrow_r t \iff \Gamma, \gamma, p \vdash rs \Rightarrow_r t$

⟨proof⟩

lemma *call-eq*: $\Gamma \ c = \text{Some } rs \implies \text{matches } \gamma \ m \ p \implies \forall r \in \text{set } rs. \text{get-action } r \neq \text{Return} \implies \Gamma, \gamma, p \vdash \langle [\text{Rule } m \ (\text{Call } c)], s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$
 ⟨proof⟩

theorem *r-eq-orig*: $\llbracket \text{all-chains (no-call-to } c) \ \Gamma \ rs; \Gamma \ c = \text{Some } rs \rrbracket \implies \Gamma, \gamma, p \vdash rs \Rightarrow_r t \iff \Gamma, \gamma, p \vdash \langle [\text{Rule } \text{MatchAny } (\text{Call } c)], \text{Undecided} \rangle \Rightarrow t$
 ⟨proof⟩

lemma *r-no-call*: $\Gamma, \gamma, p \vdash \text{Rule } \text{MatchAny } (\text{Call } c) \# rs \Rightarrow_r t \implies \Gamma \ c = \text{None} \implies \text{False}$
 ⟨proof⟩

lemma *no-call*: $\Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t \implies rs = [\text{Rule } \text{MatchAny } (\text{Call } c)] \implies s = \text{Undecided} \implies \Gamma \ c = \text{None} \implies \text{False}$

⟨proof⟩ **corollary** *r-eq-orig'*: **assumes** $\forall rs \in \text{ran } \Gamma. \text{no-call-to } c \ rs$

shows $\Gamma, \gamma, p \vdash [\text{Rule } \text{MatchAny } (\text{Call } c)] \Rightarrow_r t \iff \Gamma, \gamma, p \vdash \langle [\text{Rule } \text{MatchAny } (\text{Call } c)], \text{Undecided} \rangle \Rightarrow t$

<proof>

lemma *r-tail*: **assumes** $\Gamma, \gamma, p \vdash rs1 \Rightarrow_r$ *Decision X* **shows** $\Gamma, \gamma, p \vdash rs1 @ rs2 \Rightarrow_r$
Decision X

<proof>

lemma *r-seq*: $\Gamma, \gamma, p \vdash rs1 \Rightarrow_r$ *Undecided* $\implies \forall r \in \text{set } rs1. \neg(\text{get-action } r = \text{Return}$
 $\wedge \text{matches } \gamma (\text{get-match } r) p)$

$\implies \Gamma, \gamma, p \vdash rs2 \Rightarrow_r t \implies \Gamma, \gamma, p \vdash rs1 @ rs2 \Rightarrow_r t$

<proof>

lemma *r-appendD*: $\Gamma, \gamma, p \vdash rs1 @ rs2 \Rightarrow_r t \implies \exists s. \Gamma, \gamma, p \vdash rs1 \Rightarrow_r s$

<proof>

corollary *iptables-bigstep-r-eq*: **assumes** $\forall rs \in \text{ran } \Gamma. \text{no-call-to } c \text{ } rs \ A = \text{Accept}$
 $\vee A = \text{Drop}$

shows $\Gamma, \gamma, p \vdash [\text{Rule MatchAny } (\text{Call } c), \text{Rule MatchAny } A] \Rightarrow_r t \iff \Gamma, \gamma, p \vdash$
 $[\text{Rule MatchAny } (\text{Call } c), \text{Rule MatchAny } A], \text{Undecided} \Rightarrow t$

<proof>

lemma *ex-no-call*: *finite S* $\implies \exists c. \forall (rs :: \text{'a rule list}) \in S. \text{no-call-to } c \text{ } rs$

<proof> **lemma** *ex-no-call'*: *finite (dom Γ)* $\implies \exists c. \Gamma \ c = \text{None} \wedge (\forall (rs :: \text{'a rule$
 $\text{list}) \in (\text{ran } \Gamma). \text{no-call-to } c \text{ } rs)$

<proof>

lemma *all-chains-no-call-upd-r*: *all-chains (no-call-to c) Γ rs* $\implies (\Gamma(c \mapsto x), \gamma, p \vdash$
 $rs \Rightarrow_r t \iff \Gamma, \gamma, p \vdash rs \Rightarrow_r t$

<proof>

lemma *all-chains-no-call-upd-orig*: *all-chains (no-call-to c) Γ rs* $\implies (\Gamma(c \mapsto x), \gamma, p \vdash$
 $\langle rs, s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$

<proof>

corollary *r-eq-orig''*: **assumes** *finite (ran Γ)* **and** $\forall r \in \text{set } rs. \text{get-action } r \neq$
Return

shows $\Gamma, \gamma, p \vdash rs \Rightarrow_r t \iff \Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow t$

<proof>

end

end

theory *Semantics-Stateful*

imports *Semantics*

begin

16 Semantics Stateful

16.1 Model 1 – Curried Stateful Matcher

Processing a packet with state can be modeled as follows: The state is σ . The primitive matcher γ_σ is a curried function where the first argument is the state and it returns a stateless primitive matcher, i.e. $\gamma = \gamma_\sigma \sigma$. With this stateless primitive matcher γ , the *iptables-bigstep* semantics are executed. As entry point, the iptables built-in chains "INPUT", "OUTPUT", and "FORWARD" with their default-policy (*Accept* or *Drop* are valid for iptables) are chosen. The semantics must yield a *Decision X*. Due to the default-policy, this is always the case if the ruleset is well-formed. When a decision is made, the state σ is updated.

inductive *semantics-stateful* ::

'a ruleset \Rightarrow
 (' $\sigma \Rightarrow$ ('a, 'p) matcher) \Rightarrow — matcher, first parameter is the state
 (' $\sigma \Rightarrow$ final-decision \Rightarrow 'p \Rightarrow ' σ) \Rightarrow — state update function after firewall has decision for a packet
 ' $\sigma \Rightarrow$ — Starting state. constant
 (string \times action) \Rightarrow — The chain and default policy the firewall evaluates. For example "FORWARD", Drop
 'p list \Rightarrow — packets to be processed
 ('p \times final-decision) list \Rightarrow — packets which have been processed and their decision. ordered the same as the firewall processed them. oldest packet first
 ' $\sigma \Rightarrow$ — final state
 bool for Γ and γ_σ and state-update and σ_0 where
 — A list of packets *ps* waiting to be processed. Nothing has happened, start and final state are the same, the list of processed packets is empty.
semantics-stateful Γ γ_σ state-update σ_0 (built-in-chain, default-policy) *ps* [] σ_0 |

— Processing one packet
semantics-stateful Γ γ_σ state-update σ_0 (built-in-chain, default-policy) (p#ps)
 ps-processed $\sigma' \Rightarrow$
 $\Gamma, (\gamma_\sigma \sigma'), p \vdash \langle [Rule MatchAny (Call built-in-chain), Rule MatchAny default-policy], Undecided \rangle$
 $\Rightarrow Decision X \Rightarrow$
semantics-stateful Γ γ_σ state-update σ_0 (built-in-chain, default-policy) *ps* (ps-processed@[p, X]) (state-update $\sigma' X p$)

lemma *semantics-stateful-intro-process-one*: *semantics-stateful* Γ γ_σ state-upate σ_0 (built-in-chain, default-policy) (p#ps) ps-processed-old σ -old \Rightarrow

$\Gamma, \gamma_\sigma \sigma$ -old, $p \vdash \langle [Rule MatchAny (Call built-in-chain), Rule MatchAny default-policy], Undecided \rangle \Rightarrow Decision X \Rightarrow$
 $\sigma' = state-upate \sigma$ -old $X p \Rightarrow$
 ps-processed = ps-processed-old@[p, X] \Rightarrow

semantics-stateful $\Gamma \gamma_\sigma$ *state-upate* σ_0 (*built-in-chain*, *default-policy*) *ps*
ps-processed σ'
 ⟨*proof*⟩

lemma *semantics-stateful-intro-start*: $\sigma_0 = \sigma' \implies \text{ps-processed} = [] \implies$
semantics-stateful $\Gamma \gamma_\sigma$ *state-upate* σ_0 (*built-in-chain*, *default-policy*) *ps*
ps-processed σ'
 ⟨*proof*⟩

Example below

16.2 Model 2 – Packets Tagged with State Information

In this model, the matcher is completely stateless but packets are previously tagged with (static) stateful information.

inductive *semantics-stateful-packet-tagging* ::

'*a* *ruleset* \Rightarrow
 ('*a*, '*ptagged*) *matcher* \Rightarrow
 (' $\sigma \Rightarrow$ '*p* \Rightarrow '*ptagged*) \Rightarrow — tags the packet according to the current state before
 processing by firewall
 (' $\sigma \Rightarrow$ *final-decision* \Rightarrow '*p* \Rightarrow ' σ) \Rightarrow — state updater
 ' $\sigma \Rightarrow$ — Starting state. constant
 (*string* \times *action*) \Rightarrow
 '*p list* \Rightarrow — packets to be processed
 ('*p* \times *final-decision*) *list* \Rightarrow — packets which have been processed
 ' $\sigma \Rightarrow$ — final state
bool for Γ and γ and packet-tagger and state-update and σ_0 where
semantics-stateful-packet-tagging $\Gamma \gamma$ *packet-tagger state-upate* σ_0 (*built-in-chain*,
default-policy) *ps* [] σ_0 |

semantics-stateful-packet-tagging $\Gamma \gamma$ *packet-tagger state-upate* σ_0 (*built-in-chain*,
default-policy) (*p#ps*) *ps-processed* $\sigma' \implies$
 $\Gamma, \gamma, (\text{packet-tagger } \sigma' p) \vdash \langle [\text{Rule MatchAny (Call built-in-chain), Rule MatchAny}$
default-policy], Undecided $\rangle \Rightarrow$ *Decision X* \implies
semantics-stateful-packet-tagging $\Gamma \gamma$ *packet-tagger state-upate* σ_0 (*built-in-chain*,
default-policy) *ps* (*ps-processed*@[(*p*, *X*)] (*state-upate* $\sigma' X p$)

lemma *semantics-stateful-packet-tagging-intro-start*: $\sigma_0 = \sigma' \implies \text{ps-processed} =$
 [] \implies
semantics-stateful-packet-tagging $\Gamma \gamma$ *packet-tagger state-upate* σ_0 (*built-in-chain*,
default-policy) *ps ps-processed* σ'
 ⟨*proof*⟩

lemma *semantics-stateful-packet-tagging-intro-process-one*:
semantics-stateful-packet-tagging $\Gamma \gamma$ *packet-tagger state-upate* σ_0 (*built-in-chain*,
default-policy) (*p#ps*) *ps-processed-old* $\sigma\text{-old} \implies$
 $\Gamma, \gamma, (\text{packet-tagger } \sigma\text{-old } p) \vdash \langle [\text{Rule MatchAny (Call built-in-chain), Rule}$
MatchAny default-policy], Undecided $\rangle \Rightarrow$ *Decision X* \implies

$$\begin{aligned} \sigma' = \text{state-update } \sigma\text{-old } X \ p &\implies \\ \text{ps-processed} = \text{ps-processed-old}@[(p, X)] &\implies \\ \text{semantics-stateful-packet-tagging } \Gamma \ \gamma \ \text{packet-tagger} \ \text{state-update } \sigma_0 \ (\text{built-in-chain}, & \\ \text{default-policy}) \ \text{ps} \ \text{ps-processed} \ \sigma' & \\ \langle \text{proof} \rangle & \end{aligned}$$

lemma *semantics-bigstep-state-vs-tagged:*

assumes $\forall m::'m. \text{stateful-matcher}' \ \sigma \ m \ p = \text{stateful-matcher-tagged}' \ m \ (\text{packet-tagger}' \ \sigma \ p)$
shows $\Gamma, \text{stateful-matcher}' \ \sigma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow t \longleftrightarrow \Gamma, \text{stateful-matcher-tagged}', \text{packet-tagger}' \ \sigma \ p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow t$
 $\langle \text{proof} \rangle$

Both semantics are equal

theorem *semantics-stateful-vs-tagged:*

assumes $\forall m \ \sigma \ p. \text{stateful-matcher}' \ \sigma \ m \ p = \text{stateful-matcher-tagged}' \ m \ (\text{packet-tagger}' \ \sigma \ p)$
shows $\text{semantics-stateful } rs \ \text{stateful-matcher}' \ \text{state-update}' \ \sigma_0 \ \text{start } ps \ \text{ps-processed} \ \sigma' = \text{semantics-stateful-packet-tagging } rs \ \text{stateful-matcher-tagged}' \ \text{packet-tagger}' \ \text{state-update}' \ \sigma_0 \ \text{start } ps \ \text{ps-processed} \ \sigma'$
 $\langle \text{proof} \rangle$

Examples

context

begin

16.3 Example: Conntrack with curried matcher

We illustrate stateful semantics with a simple example. We allow matching on the states *New* and *Established*. In addition, we introduce a primitive *match* to match on outgoing ssh packets (*dst port = 22*). The state is managed in a state table where accepted connections are remembered.

SomePacket with source and destination port or something we don't know about

private datatype *packet* = *SomePacket* *nat* × *nat* | *OtherPacket*

private datatype *primitive-matches* = *New* | *Established* | *IsSSH*

In the state, we remember the packets which belong to an established connection.

private datatype *conntrack-state* = *State* *packet* *set*

The stateful primitive matcher: It is given the current state table. If *match* on *Established*, the packet must be known in the state table. If *match* on

New, the packet must not be in the state table. If match on *IsSSH*, the dst port of the packet must be 22.

```
private fun stateful-matcher :: contrack-state ⇒ (primitive-matches, packet)
matcher where
  stateful-matcher (State state-table) = (λm p. m = Established ∧ p ∈ state-table
  ∨
    m = New ∧ p ∉ state-table ∨
    m = IsSSH ∧ (∃ dst-port. p = SomePacket (22,
dst-port)))
```

Connections are always bi-directional.

```
private fun reverse-direction :: packet ⇒ packet where
  reverse-direction OtherPacket = OtherPacket |
  reverse-direction (SomePacket (src, dst)) = SomePacket (dst,src)
```

If a packet is accepted, the state for its bi-directional connection is saved in the state table.

```
private fun state-update' :: contrack-state ⇒ final-decision ⇒ packet ⇒ con-
ntrack-state where
  state-update' (State state-table) FinalAllow p = State (state-table ∪ {p, re-
verse-direction p}) |
  state-update' (State state-table) FinalDeny p = State state-table
```

Allow everything that is established and allow new ssh connections. Drop everything else (default policy, see below)

```
private definition ruleset == ["INPUT" ↦ [Rule (Match Established) Accept,
Rule (MatchAnd (Match IsSSH) (Match New)) Accept]]
```

The *ruleset* does not allow *OtherPacket*

```
lemma semantics-stateful ruleset stateful-matcher state-update' (State {}) ("INPUT",
Drop) []
  [(OtherPacket, FinalDeny)] (State {})
  ⟨proof⟩
```

The *ruleset* allows ssh packets, i.e. any packets with destination port 22 in the *New* rule. The state is updated such that everything which belongs to the connection will now be accepted.

```
lemma semantics-stateful ruleset stateful-matcher state-update' (State {}) ("INPUT",
Drop)
  []
  [(SomePacket (22, 1024), FinalAllow)]
  (State {SomePacket (1024, 22), SomePacket (22, 1024)})
  ⟨proof⟩
```

If we continue with this state, answer packets are now allowed

```
lemma semantics-stateful ruleset stateful-matcher state-update' (State {}) ("INPUT",
Drop)
```



```

    []
    [(SomePacket (22, 1024), FinalAllow), (SomePacket (1024, 22), FinalAl-
low)]
    (State {SomePacket (1024, 22), SomePacket (22, 1024)})
    <proof>

```

In contrast, without having previously established a state, answer packets are prohibited

If we continue with this state, answer packets are now allowed

lemma *semantics-stateful ruleset stateful-matcher state-update'* (State {}) ("INPUT", Drop)

```

    []
    [(SomePacket (1024, 22), FinalDeny), (SomePacket (22, 1024), FinalAl-
low), (SomePacket (1024, 22), FinalAllow)]
    (State {SomePacket (1024, 22), SomePacket (22, 1024)})
    <proof>

```

16.4 Example: Contrack with packet tagging

datatype *packet-tag* = TagNew | TagEstablished

datatype *packet-tagged* = SomePacket-tagged nat × nat × packet-tag | OtherPacket-tagged packet-tag

fun *get-packet-tag* :: packet-tagged ⇒ packet-tag **where**

```

  get-packet-tag (SomePacket-tagged (-,-, tag)) = tag |
  get-packet-tag (OtherPacket-tagged tag) = tag

```

definition *stateful-matcher-tagged* :: (primitive-matches, packet-tagged) matcher **where**

stateful-matcher-tagged ≡ λm p. m = Established ∧ (get-packet-tag p = TagEstablished) ∨

$$m = New \wedge (get-packet-tag p = TagNew) \vee$$

$$m = IsSSH \wedge (\exists dst-port tag. p = SomePacket-tagged$$

(22, dst-port, tag))

fun *calculate-packet-tag* :: contrack-state ⇒ packet ⇒ packet-tag **where**

calculate-packet-tag (State state-table) p = (if p ∈ state-table then TagEstablished else TagNew)

fun *packet-tagger* :: contrack-state ⇒ packet ⇒ packet-tagged **where**

```

  packet-tagger σ (SomePacket (s,d)) = (SomePacket-tagged (s,d, calculate-packet-tag
σ (SomePacket (s,d)))) |
  packet-tagger σ OtherPacket = (OtherPacket-tagged (calculate-packet-tag σ
OtherPacket))

```

If a packet is accepted, the state for its bi-directional connection is saved in the state table.

fun *state-update-tagged* :: contrack-state ⇒ final-decision ⇒ packet ⇒ contrack-state **where**

```

    state-update-tagged (State state-table) FinalAllow p = State (state-table  $\cup$  {p,
reverse-direction p}) |
    state-update-tagged (State state-table) FinalDeny p = State state-table

```

Both semantics are equal

```

lemma semantics-stateful rs stateful-matcher state-update'  $\sigma_0$  start ps ps-processed
 $\sigma' =$ 
    semantics-stateful-packet-tagging rs stateful-matcher-tagged packet-tagger state-update'
 $\sigma_0$  start ps ps-processed  $\sigma'$ 
    <proof>
end

```

end

theory Semantics-Goto

```

imports Main Firewall-Common Common/List-Misc HOL-Library.LaTeXsugar
begin

```

17 Big Step Semantics with Goto

We extend the iptables semantics to support the goto action. A goto directly continues processing at the start of the called chain. It does not change the call stack. In contrast to calls, goto does not return. Consequently, everything behind a matching goto cannot be reached.

This theory is structured as follows. First, the goto semantics are introduced. Then, we show that those semantics are deterministic. Finally, we present two methods to remove gotos. The first unfolds goto. The second replaces gotos with calls. Finally, since the goto rules makes all proofs quite ugly, we never mention the goto semantics again. As we have shown, we can get rid of the gotos easily, thus, we stick to the nicer iptables semantics without goto.

```

context
begin

```

17.1 Semantics

```

private type-synonym 'a ruleset = string  $\rightarrow$  'a rule list

```

```

private type-synonym ('a, 'p) matcher = 'a  $\Rightarrow$  'p  $\Rightarrow$  bool

```

```

qualified fun matches :: ('a, 'p) matcher  $\Rightarrow$  'a match-expr  $\Rightarrow$  'p  $\Rightarrow$  bool where
    matches  $\gamma$  (MatchAnd e1 e2) p  $\longleftrightarrow$  matches  $\gamma$  e1 p  $\wedge$  matches  $\gamma$  e2 p |
    matches  $\gamma$  (MatchNot me) p  $\longleftrightarrow$   $\neg$  matches  $\gamma$  me p |
    matches  $\gamma$  (Match e) p  $\longleftrightarrow$   $\gamma$  e p |
    matches - MatchAny -  $\longleftrightarrow$  True

```

qualified fun *no-matching-Goto* :: ('a, 'p) matcher ⇒ 'p ⇒ 'a rule list ⇒ bool
where

```

no-matching-Goto - - [] ←→ True |
no-matching-Goto γ p ((Rule m (Goto -))#rs) ←→ ¬ matches γ m p ∧
no-matching-Goto γ p rs |
no-matching-Goto γ p (-#rs) ←→ no-matching-Goto γ p rs

```

inductive *iptables-goto-bigstep* :: 'a ruleset ⇒ ('a, 'p) matcher ⇒ 'p ⇒ 'a rule list ⇒ state ⇒ state ⇒ bool

```

(⟨-, -, ⟦_⟧, ⟦_⟧⟩ ⇒ -) [60,60,60,20,98,98] 89)
for Γ and γ and p where
skip:   Γ, γ, p ⊢_g ⟨[], t⟩ ⇒ t |
accept: matches γ m p ⇒ Γ, γ, p ⊢_g ⟨[Rule m Accept], Undecided⟩ ⇒ Decision
FinalAllow |
drop:   matches γ m p ⇒ Γ, γ, p ⊢_g ⟨[Rule m Drop], Undecided⟩ ⇒ Decision
FinalDeny |
reject: matches γ m p ⇒ Γ, γ, p ⊢_g ⟨[Rule m Reject], Undecided⟩ ⇒ Decision
FinalDeny |
log:    matches γ m p ⇒ Γ, γ, p ⊢_g ⟨[Rule m Log], Undecided⟩ ⇒ Undecided |
empty:  matches γ m p ⇒ Γ, γ, p ⊢_g ⟨[Rule m Empty], Undecided⟩ ⇒ Undecided
|
nomatch: ¬ matches γ m p ⇒ Γ, γ, p ⊢_g ⟨[Rule m a], Undecided⟩ ⇒ Undecided
|
decision: Γ, γ, p ⊢_g ⟨rs, Decision X⟩ ⇒ Decision X |
seq:     [[Γ, γ, p ⊢_g ⟨rs1, Undecided⟩ ⇒ t; Γ, γ, p ⊢_g ⟨rs2, t⟩ ⇒ t'; no-matching-Goto
γ p rs1]] ⇒ Γ, γ, p ⊢_g ⟨rs1@rs2, Undecided⟩ ⇒ t' |
call-return: [[ matches γ m p; Γ chain = Some (rs1@[Rule m' Return]@rs2);
matches γ m' p; Γ, γ, p ⊢_g ⟨rs1, Undecided⟩ ⇒ Undecided;
no-matching-Goto γ p rs1]] ⇒
— we do not support a goto in the first part if you want to return
— probably unhandled case:
— main:
—   call foo
— foo:
—   goto bar
— bar:
—   Return //returns to call foo
— But this would be a really awkward ruleset!
Γ, γ, p ⊢_g ⟨[Rule m (Call chain)], Undecided⟩ ⇒ Undecided |
call-result: [[ matches γ m p; Γ chain = Some rs; Γ, γ, p ⊢_g ⟨rs, Undecided⟩ ⇒ t
]] ⇒
Γ, γ, p ⊢_g ⟨[Rule m (Call chain)], Undecided⟩ ⇒ t | — goto handling
here seems okay
goto-decision: [[ matches γ m p; Γ chain = Some rs; Γ, γ, p ⊢_g ⟨rs, Undecided⟩
⇒ Decision X ]] ⇒
Γ, γ, p ⊢_g ⟨(Rule m (Goto chain))#rest, Undecided⟩ ⇒ Decision X |
goto-no-decision: [[ matches γ m p; Γ chain = Some rs; Γ, γ, p ⊢_g ⟨rs, Undecided⟩
⇒ Undecided ]] ⇒

```

$$\Gamma, \gamma, p \vdash_g \langle (Rule\ m\ (Goto\ chain)) \# rest, Undecided \rangle \Rightarrow Undecided$$

The semantic rules again in pretty format:

$$\begin{array}{c}
\frac{}{\Gamma, \gamma, p \vdash_g \langle [], t \rangle \Rightarrow t} \\
\frac{matches\ \gamma\ m\ p}{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ Accept], Undecided \rangle \Rightarrow Decision\ FinalAllow} \\
\frac{matches\ \gamma\ m\ p}{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ Drop], Undecided \rangle \Rightarrow Decision\ FinalDeny} \\
\frac{matches\ \gamma\ m\ p}{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ Reject], Undecided \rangle \Rightarrow Decision\ FinalDeny} \\
\frac{matches\ \gamma\ m\ p}{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ Log], Undecided \rangle \Rightarrow Undecided} \\
\frac{\neg\ matches\ \gamma\ m\ p}{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ Empty], Undecided \rangle \Rightarrow Undecided} \\
\frac{\Gamma, \gamma, p \vdash_g \langle rs, Decision\ X \rangle \Rightarrow Decision\ X}{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ a], Undecided \rangle \Rightarrow Undecided} \\
\frac{\Gamma, \gamma, p \vdash_g \langle rs_1, Undecided \rangle \Rightarrow t \quad \Gamma, \gamma, p \vdash_g \langle rs_2, t \rangle \Rightarrow t' \quad no\text{-}matching\text{-}Goto\ \gamma\ p\ rs_1}{\Gamma, \gamma, p \vdash_g \langle rs_1 @ rs_2, Undecided \rangle \Rightarrow t'} \\
\frac{matches\ \gamma\ m\ p \quad \Gamma\ chain = Some\ (rs_1 @ [Rule\ m' Return] @ rs_2) \quad matches\ \gamma\ m' p}{\Gamma, \gamma, p \vdash_g \langle rs_1, Undecided \rangle \Rightarrow Undecided \quad no\text{-}matching\text{-}Goto\ \gamma\ p\ rs_1} \\
\frac{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ (Call\ chain)], Undecided \rangle \Rightarrow Undecided}{matches\ \gamma\ m\ p \quad \Gamma\ chain = Some\ rs \quad \Gamma, \gamma, p \vdash_g \langle rs, Undecided \rangle \Rightarrow t} \\
\frac{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ (Call\ chain)], Undecided \rangle \Rightarrow t}{matches\ \gamma\ m\ p} \\
\frac{\Gamma\ chain = Some\ rs \quad \Gamma, \gamma, p \vdash_g \langle rs, Undecided \rangle \Rightarrow Decision\ X}{\Gamma, \gamma, p \vdash_g \langle Rule\ m\ (Goto\ chain) \cdot rest, Undecided \rangle \Rightarrow Decision\ X} \\
\frac{matches\ \gamma\ m\ p \quad \Gamma\ chain = Some\ rs \quad \Gamma, \gamma, p \vdash_g \langle rs, Undecided \rangle \Rightarrow Undecided}{\Gamma, \gamma, p \vdash_g \langle Rule\ m\ (Goto\ chain) \cdot rest, Undecided \rangle \Rightarrow Undecided}
\end{array}$$

private lemma deny:

$$matches\ \gamma\ m\ p \Longrightarrow a = Drop \vee a = Reject \Longrightarrow iptables\text{-}goto\text{-}bigstep\ \Gamma\ \gamma\ p\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalDeny)$$

⟨proof⟩ **lemma** *iptables-goto-bigstep-induct*
 [case-names
 Skip Allow Deny Log Nomatch Decision Seq Call-return Call-result Goto-Decision
 Goto-no-Decision,
 induct pred: *iptables-goto-bigstep*]:
 $\llbracket \Gamma, \gamma, p \vdash_g \langle rs, s \rangle \Rightarrow t;$
 $\bigwedge t. P \llbracket t t;$
 $\bigwedge m a. \text{matches } \gamma m p \Rightarrow a = \text{Accept} \Rightarrow P [\text{Rule } m a] \text{ Undecided (Decision FinalAllow)};$
 $\bigwedge m a. \text{matches } \gamma m p \Rightarrow a = \text{Drop} \vee a = \text{Reject} \Rightarrow P [\text{Rule } m a] \text{ Undecided (Decision FinalDeny)};$
 $\bigwedge m a. \text{matches } \gamma m p \Rightarrow a = \text{Log} \vee a = \text{Empty} \Rightarrow P [\text{Rule } m a] \text{ Undecided Undecided};$
 $\bigwedge m a. \neg \text{matches } \gamma m p \Rightarrow P [\text{Rule } m a] \text{ Undecided Undecided};$
 $\bigwedge rs X. P rs (\text{Decision } X) (\text{Decision } X);$
 $\bigwedge rs rs_1 rs_2 t t'. rs = rs_1 @ rs_2 \Rightarrow \Gamma, \gamma, p \vdash_g \langle rs_1, \text{Undecided} \rangle \Rightarrow t \Rightarrow P rs_1 \text{ Undecided } t \Rightarrow$
 $\Gamma, \gamma, p \vdash_g \langle rs_2, t \rangle \Rightarrow t' \Rightarrow P rs_2 t t' \Rightarrow \text{no-matching-Goto } \gamma p rs_1 \Rightarrow$
 $P rs \text{ Undecided } t';$
 $\bigwedge m a \text{ chain } rs_1 m' rs_2. \text{matches } \gamma m p \Rightarrow a = \text{Call chain} \Rightarrow$
 $\Gamma \text{ chain} = \text{Some } (rs_1 @ [\text{Rule } m' \text{Return}] @ rs_2) \Rightarrow$
 $\text{matches } \gamma m' p \Rightarrow \Gamma, \gamma, p \vdash_g \langle rs_1, \text{Undecided} \rangle \Rightarrow \text{Undecided}$
 \Rightarrow
 $\text{no-matching-Goto } \gamma p rs_1 \Rightarrow P rs_1 \text{ Undecided Undecided}$
 \Rightarrow
 $P [\text{Rule } m a] \text{ Undecided Undecided};$
 $\bigwedge m a \text{ chain } rs t. \text{matches } \gamma m p \Rightarrow a = \text{Call chain} \Rightarrow \Gamma \text{ chain} = \text{Some } rs \Rightarrow$
 $\Gamma, \gamma, p \vdash_g \langle rs, \text{Undecided} \rangle \Rightarrow t \Rightarrow P rs \text{ Undecided } t \Rightarrow P [\text{Rule } m a] \text{ Undecided } t;$
 $\bigwedge m a \text{ chain } rs \text{ rest } X. \text{matches } \gamma m p \Rightarrow a = \text{Goto chain} \Rightarrow \Gamma \text{ chain} = \text{Some } rs \Rightarrow$
 $\Gamma, \gamma, p \vdash_g \langle rs, \text{Undecided} \rangle \Rightarrow (\text{Decision } X) \Rightarrow P rs \text{ Undecided (Decision } X) \Rightarrow$
 $P (\text{Rule } m a \# \text{rest}) \text{ Undecided (Decision } X);$
 $\bigwedge m a \text{ chain } rs \text{ rest}. \text{matches } \gamma m p \Rightarrow a = \text{Goto chain} \Rightarrow \Gamma \text{ chain} = \text{Some } rs \Rightarrow$
 $\Gamma, \gamma, p \vdash_g \langle rs, \text{Undecided} \rangle \Rightarrow \text{Undecided} \Rightarrow P rs \text{ Undecided Undecided} \Rightarrow$
 $P (\text{Rule } m a \# \text{rest}) \text{ Undecided Undecided} \rrbracket \Rightarrow$
 $P rs s t$
 ⟨proof⟩

17.1.1 Forward reasoning

private lemma *decisionD*: $\Gamma, \gamma, p \vdash_g \langle r, s \rangle \Rightarrow t \Rightarrow s = \text{Decision } X \Rightarrow t = \text{Decision } X$

⟨proof⟩ **lemma** *iptables-goto-bigstep-to-undecided*: $\Gamma, \gamma, p \vdash_g \langle rs, s \rangle \Rightarrow \text{Undecided}$

$\implies s = \text{Undecided}$
 ⟨proof⟩ **lemma** *iptables-goto-bigstep-to-decision*: $\Gamma, \gamma, p \vdash_g \langle rs, \text{Decision } Y \rangle \Rightarrow$
Decision } X \implies Y = X
 ⟨proof⟩ **lemma** *skipD*: $\Gamma, \gamma, p \vdash_g \langle r, s \rangle \Rightarrow t \implies r = [] \implies s = t$
 ⟨proof⟩ **lemma** *gotoD*: $\Gamma, \gamma, p \vdash_g \langle r, s \rangle \Rightarrow t \implies r = [\text{Rule } m \text{ (Goto chain)}]$
 $\implies s = \text{Undecided} \implies \text{matches } \gamma \text{ } m \text{ } p \implies$
 $\exists rs. \Gamma \text{ chain} = \text{Some } rs \wedge \Gamma, \gamma, p \vdash_g \langle rs, s \rangle \Rightarrow t$
 ⟨proof⟩ **lemma** *not-no-matching-Goto-singleton-cases*: $\neg \text{no-matching-Goto } \gamma$
 $p \text{ [Rule } m \text{ } a] \iff (\exists \text{ chain. } a = (\text{Goto chain})) \wedge \text{matches } \gamma \text{ } m \text{ } p$
 ⟨proof⟩ **lemma** *no-matching-Goto-Cons*: *no-matching-Goto } \gamma \text{ } p \text{ [r] } \implies*
no-matching-Goto } \gamma \text{ } p \text{ } rs \implies \text{no-matching-Goto } \gamma \text{ } p \text{ (r\#rs)}
 ⟨proof⟩ **lemma** *no-matching-Goto-head*: *no-matching-Goto } \gamma \text{ } p \text{ (r\#rs) } \implies*
no-matching-Goto } \gamma \text{ } p \text{ [r]
 ⟨proof⟩ **lemma** *no-matching-Goto-tail*: *no-matching-Goto } \gamma \text{ } p \text{ (r\#rs) } \implies*
no-matching-Goto } \gamma \text{ } p \text{ } rs
 ⟨proof⟩ **lemma** *not-no-matching-Goto-cases*:
assumes $\neg \text{no-matching-Goto } \gamma \text{ } p \text{ } rs \text{ } rs \neq []$
shows $\exists rs1 \text{ } m \text{ chain } rs2. rs = rs1 @ (\text{Rule } m \text{ (Goto chain)}) \# rs2 \wedge \text{no-matching-Goto}$
 $\gamma \text{ } p \text{ } rs1 \wedge \text{matches } \gamma \text{ } m \text{ } p$
 ⟨proof⟩ **lemma** *seq-cons-Goto-Undecided*:
assumes $\Gamma, \gamma, p \vdash_g \langle [\text{Rule } m \text{ (Goto chain)}], \text{Undecided} \rangle \Rightarrow \text{Undecided}$
and $\neg \text{matches } \gamma \text{ } m \text{ } p \implies \Gamma, \gamma, p \vdash_g \langle rs, \text{Undecided} \rangle \Rightarrow \text{Undecided}$
shows $\Gamma, \gamma, p \vdash_g \langle \text{Rule } m \text{ (Goto chain)} \# rs, \text{Undecided} \rangle \Rightarrow \text{Undecided}$
 ⟨proof⟩ **lemma** *seq-cons-Goto-t*:
 $\Gamma, \gamma, p \vdash_g \langle [\text{Rule } m \text{ (Goto chain)}], \text{Undecided} \rangle \Rightarrow t \implies \text{matches } \gamma \text{ } m \text{ } p \implies$
 $\Gamma, \gamma, p \vdash_g \langle \text{Rule } m \text{ (Goto chain)} \# rs, \text{Undecided} \rangle \Rightarrow t$
 ⟨proof⟩ **lemma** *no-matching-Goto-append*: *no-matching-Goto } \gamma \text{ } p \text{ (rs1@rs2)*
 $\iff \text{no-matching-Goto } \gamma \text{ } p \text{ } rs1 \wedge \text{no-matching-Goto } \gamma \text{ } p \text{ } rs2$
 ⟨proof⟩ **lemma** *no-matching-Goto-append1*: *no-matching-Goto } \gamma \text{ } p \text{ (rs1@rs2)*
 $\implies \text{no-matching-Goto } \gamma \text{ } p \text{ } rs1$
 ⟨proof⟩ **lemma** *no-matching-Goto-append2*: *no-matching-Goto } \gamma \text{ } p \text{ (rs1@rs2)*
 $\implies \text{no-matching-Goto } \gamma \text{ } p \text{ } rs2$
 ⟨proof⟩ **lemma** *seq-cons*:
assumes $\Gamma, \gamma, p \vdash_g \langle [r], \text{Undecided} \rangle \Rightarrow t$ **and** $\Gamma, \gamma, p \vdash_g \langle rs, t \rangle \Rightarrow t'$ **and** *no-matching-Goto*
 $\gamma \text{ } p \text{ [r]}$
shows $\Gamma, \gamma, p \vdash_g \langle r \# rs, \text{Undecided} \rangle \Rightarrow t'$
 ⟨proof⟩

context
notes *skipD[dest] list-app-singletonE[elim]*
begin
lemma *acceptD*: $\Gamma, \gamma, p \vdash_g \langle r, s \rangle \Rightarrow t \implies r = [\text{Rule } m \text{ Accept}] \implies \text{matches } \gamma$
 $m \text{ } p \implies s = \text{Undecided} \implies t = \text{Decision FinalAllow}$
 ⟨proof⟩

lemma *dropD*: $\Gamma, \gamma, p \vdash_g \langle r, s \rangle \Rightarrow t \implies r = [\text{Rule } m \text{ Drop}] \implies \text{matches } \gamma \text{ } m$
 $p \implies s = \text{Undecided} \implies t = \text{Decision FinalDeny}$
 ⟨proof⟩

lemma rejectD: $\Gamma, \gamma, p \vdash_g \langle r, s \rangle \Rightarrow t \Longrightarrow r = [\text{Rule } m \text{ Reject}] \Longrightarrow \text{matches } \gamma$
 $m \ p \Longrightarrow s = \text{Undecided} \Longrightarrow t = \text{Decision FinalDeny}$
 ⟨proof⟩

lemma logD: $\Gamma, \gamma, p \vdash_g \langle r, s \rangle \Rightarrow t \Longrightarrow r = [\text{Rule } m \text{ Log}] \Longrightarrow \text{matches } \gamma \ m \ p$
 $\Longrightarrow s = \text{Undecided} \Longrightarrow t = \text{Undecided}$
 ⟨proof⟩

lemma emptyD: $\Gamma, \gamma, p \vdash_g \langle r, s \rangle \Rightarrow t \Longrightarrow r = [\text{Rule } m \text{ Empty}] \Longrightarrow \text{matches } \gamma$
 $m \ p \Longrightarrow s = \text{Undecided} \Longrightarrow t = \text{Undecided}$
 ⟨proof⟩

lemma nomatchD: $\Gamma, \gamma, p \vdash_g \langle r, s \rangle \Rightarrow t \Longrightarrow r = [\text{Rule } m \ a] \Longrightarrow s = \text{Undecided}$
 $\Longrightarrow \neg \text{matches } \gamma \ m \ p \Longrightarrow t = \text{Undecided}$
 ⟨proof⟩

lemma callD:

assumes $\Gamma, \gamma, p \vdash_g \langle r, s \rangle \Rightarrow t \ r = [\text{Rule } m \ (\text{Call chain})] \ s = \text{Undecided}$
matches $\gamma \ m \ p \ \Gamma \ \text{chain} = \text{Some } rs$

obtains $\Gamma, \gamma, p \vdash_g \langle rs, s \rangle \Rightarrow t$

| $rs_1 \ rs_2 \ m' \ \text{where } rs = rs_1 \ @ \ \text{Rule } m' \ \text{Return } \# \ rs_2 \ \text{matches } \gamma \ m' \ p$
 $\Gamma, \gamma, p \vdash_g \langle rs_1, s \rangle \Rightarrow \text{Undecided no-matching-Goto } \gamma \ p \ rs_1 \ t = \text{Undecided}$

⟨proof⟩

end

private lemmas *iptables-goto-bigstepD* = *skipD* *acceptD* *dropD* *rejectD* *logD*
emptyD *nomatchD* *decisionD* *callD* *gotoD*

private lemma *seq'*:

assumes $rs = rs_1 \ @ \ rs_2 \ \Gamma, \gamma, p \vdash_g \langle rs_1, s \rangle \Rightarrow t \ \Gamma, \gamma, p \vdash_g \langle rs_2, t \rangle \Rightarrow t'$ **and**
no-matching-Goto $\gamma \ p \ rs_1$

shows $\Gamma, \gamma, p \vdash_g \langle rs, s \rangle \Rightarrow t'$

⟨proof⟩ **lemma** *seq'-cons:* $\Gamma, \gamma, p \vdash_g \langle [r], s \rangle \Rightarrow t \Longrightarrow \Gamma, \gamma, p \vdash_g \langle rs, t \rangle \Rightarrow t' \Longrightarrow$
no-matching-Goto $\gamma \ p \ [r] \Longrightarrow \Gamma, \gamma, p \vdash_g \langle r \# rs, s \rangle \Rightarrow t'$

⟨proof⟩ **lemma** *no-matching-Goto-take:* *no-matching-Goto* $\gamma \ p \ rs \Longrightarrow \text{no-matching-Goto}$
 $\gamma \ p \ (\text{take } n \ rs)$

⟨proof⟩ **lemma** *seq-split:*

assumes $\Gamma, \gamma, p \vdash_g \langle rs, s \rangle \Rightarrow t \ rs = rs_1 \ @ \ rs_2$

obtains (*no-matching-Goto*) t' **where** $\Gamma, \gamma, p \vdash_g \langle rs_1, s \rangle \Rightarrow t' \ \Gamma, \gamma, p \vdash_g \langle rs_2, t' \rangle$
 $\Rightarrow t \ \text{no-matching-Goto } \gamma \ p \ rs_1$

| (*matching-Goto*) $\Gamma, \gamma, p \vdash_g \langle rs_1, s \rangle \Rightarrow t \ \neg \ \text{no-matching-Goto } \gamma \ p \ rs_1$

⟨proof⟩ **lemma** *seqE:*

assumes $\Gamma, \gamma, p \vdash_g \langle rs_1 \ @ \ rs_2, s \rangle \Rightarrow t$

obtains (*no-matching-Goto*) ti **where** $\Gamma, \gamma, p \vdash_g \langle rs_1, s \rangle \Rightarrow ti \ \Gamma, \gamma, p \vdash_g \langle rs_2, ti \rangle$
 $\Rightarrow t \ \text{no-matching-Goto } \gamma \ p \ rs_1$

| (*matching-Goto*) $\Gamma, \gamma, p \vdash_g \langle rs_1, s \rangle \Rightarrow t \ \neg \ \text{no-matching-Goto } \gamma \ p \ rs_1$

⟨proof⟩ **lemma** *seqE-cons:*

assumes $\Gamma, \gamma, p \vdash_g \langle r \# rs, s \rangle \Rightarrow t$

obtains (*no-matching-Goto*) ti **where** $\Gamma, \gamma, p \vdash_g \langle [r], s \rangle \Rightarrow ti \ \Gamma, \gamma, p \vdash_g \langle rs, ti \rangle \Rightarrow$

t *no-matching-Goto* γ p $[r]$
 $|$ (*matching-Goto*) $\Gamma, \gamma, p \vdash_g \langle [r], s \rangle \Rightarrow t \neg$ *no-matching-Goto* γ p $[r]$
 \langle proof \rangle **lemma** *seqE-cons-Undecided*:
assumes $\Gamma, \gamma, p \vdash_g \langle r \# rs, \text{Undecided} \rangle \Rightarrow t$
obtains (*no-matching-Goto*) ti **where** $\Gamma, \gamma, p \vdash_g \langle [r], \text{Undecided} \rangle \Rightarrow ti$ **and**
 $\Gamma, \gamma, p \vdash_g \langle rs, ti \rangle \Rightarrow t$ **and** *no-matching-Goto* γ p $[r]$
 $|$ (*matching-Goto*) m *chain* rs' **where** $r = \text{Rule } m$ (*Goto chain*) **and**
 $\Gamma, \gamma, p \vdash_g \langle [\text{Rule } m$ (*Goto chain*)], $\text{Undecided} \rangle \Rightarrow t$ **and** *matches* γ m p Γ *chain* =
Some rs'
 \langle proof \rangle **lemma** *nomatch'*:
assumes $\bigwedge r. r \in \text{set } rs \Longrightarrow \neg$ *matches* γ (*get-match* r) p
shows $\Gamma, \gamma, p \vdash_g \langle rs, s \rangle \Rightarrow s$
 \langle proof \rangle **lemma** *no-free-return*: **assumes** $\Gamma, \gamma, p \vdash_g \langle [\text{Rule } m$ *Return*], $\text{Undecided} \rangle$
 $\Rightarrow t$ **and** *matches* γ m p **shows** *False*
 \langle proof \rangle

17.2 Determinism

private lemma *iptables-goto-bigstep-Undecided-Undecided-deterministic*:
 $\Gamma, \gamma, p \vdash_g \langle rs, \text{Undecided} \rangle \Rightarrow \text{Undecided} \Longrightarrow \Gamma, \gamma, p \vdash_g \langle rs, \text{Undecided} \rangle \Rightarrow t \Longrightarrow$
 $t = \text{Undecided}$
 \langle proof \rangle **lemma** *iptables-goto-bigstep-Undecided-deterministic*:
 $\Gamma, \gamma, p \vdash_g \langle rs, \text{Undecided} \rangle \Rightarrow t \Longrightarrow \Gamma, \gamma, p \vdash_g \langle rs, \text{Undecided} \rangle \Rightarrow t' \Longrightarrow t' = t$
 \langle proof \rangle **theorem** *iptables-goto-bigstep-deterministic*: **assumes** $\Gamma, \gamma, p \vdash_g \langle rs, s \rangle$
 $\Rightarrow t$ **and** $\Gamma, \gamma, p \vdash_g \langle rs, s \rangle \Rightarrow t'$ **shows** $t = t'$
 \langle proof \rangle

17.3 Matching

private lemma *matches-rule-and-simp-help*:
assumes *matches* γ m p
shows $\Gamma, \gamma, p \vdash_g \langle [\text{Rule}$ (*MatchAnd* m m') a], $s \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, p \vdash_g \langle [\text{Rule } m'$
 a], $s \rangle \Rightarrow t$ (**is** $?l \longleftrightarrow ?r$)
 \langle proof \rangle **lemma** *matches-MatchNot-simp*:
assumes *matches* γ m p
shows $\Gamma, \gamma, p \vdash_g \langle [\text{Rule}$ (*MatchNot* m) a], $\text{Undecided} \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, p \vdash_g \langle [],$
 $\text{Undecided} \rangle \Rightarrow t$ (**is** $?l \longleftrightarrow ?r$)
 \langle proof \rangle **lemma** *matches-MatchNotAnd-simp*:
assumes *matches* γ m p
shows $\Gamma, \gamma, p \vdash_g \langle [\text{Rule}$ (*MatchAnd* (*MatchNot* m) m') a], $\text{Undecided} \rangle \Rightarrow t \longleftrightarrow$
 $\Gamma, \gamma, p \vdash_g \langle [], \text{Undecided} \rangle \Rightarrow t$ (**is** $?l \longleftrightarrow ?r$)
 \langle proof \rangle **lemma** *matches-rule-and-simp*:
assumes *matches* γ m p
shows $\Gamma, \gamma, p \vdash_g \langle [\text{Rule}$ (*MatchAnd* m m') a], $s \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, p \vdash_g \langle [\text{Rule } m'$
 a], $s \rangle \Rightarrow t$
 \langle proof \rangle **definition** *add-match* $:: 'a$ *match-expr* $\Rightarrow 'a$ *rule list* $\Rightarrow 'a$ *rule list*
where
 $\text{add-match } m$ $rs = \text{map } (\lambda r. \text{case } r \text{ of Rule } m' a' \Rightarrow \text{Rule} (\text{MatchAnd } m m')$
 $a')$ rs

private lemma *add-match-split*: $\text{add-match } m \text{ (rs1@rs2)} = \text{add-match } m \text{ rs1}$
@ *add-match m rs2*
⟨proof⟩ **lemma** *add-match-split-fst*: $\text{add-match } m \text{ (Rule } m' \text{ a' \# rs)} = \text{Rule}$
(*MatchAnd m m'*) *a' \# add-match m rs*
⟨proof⟩ **lemma** *matches-add-match-no-matching-Goto-simp*: $\text{matches } \gamma \text{ m p}$
 $\implies \text{no-matching-Goto } \gamma \text{ p (add-match m rs)} \implies \text{no-matching-Goto } \gamma \text{ p rs}$
⟨proof⟩ **lemma** *matches-add-match-no-matching-Goto-simp2*: $\text{matches } \gamma \text{ m p}$
 $\implies \text{no-matching-Goto } \gamma \text{ p rs} \implies \text{no-matching-Goto } \gamma \text{ p (add-match m rs)}$
⟨proof⟩ **lemma** *matches-add-match-MatchNot-no-matching-Goto-simp*: \neg
 $\text{matches } \gamma \text{ m p} \implies \text{no-matching-Goto } \gamma \text{ p (add-match m rs)}$
⟨proof⟩ **lemma** *not-matches-add-match-simp*:
assumes $\neg \text{matches } \gamma \text{ m p}$
shows $\Gamma, \gamma, \text{pl}_g \langle \text{add-match } m \text{ rs, Undecided} \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, \text{pl}_g \langle [], \text{Undecided} \rangle$
 $\Rightarrow t$
⟨proof⟩ **lemma** *matches-add-match-MatchNot-simp*:
assumes $m: \text{matches } \gamma \text{ m p}$
shows $\Gamma, \gamma, \text{pl}_g \langle \text{add-match (MatchNot m) rs, s} \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, \text{pl}_g \langle [], s \rangle \Rightarrow t$
(is ?l s \longleftrightarrow ?r s)
⟨proof⟩ **lemma** *just-show-all-bigstep-semantic-equalities-with-start-Undecided*:
 $\Gamma, \gamma, \text{pl}_g \langle \text{rs1, Undecided} \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, \text{pl}_g \langle \text{rs2, Undecided} \rangle \Rightarrow t \implies$
 $\Gamma, \gamma, \text{pl}_g \langle \text{rs1, s} \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, \text{pl}_g \langle \text{rs2, s} \rangle \Rightarrow t$
⟨proof⟩ **lemma** *matches-add-match-simp-helper*:
assumes $m: \text{matches } \gamma \text{ m p}$
shows $\Gamma, \gamma, \text{pl}_g \langle \text{add-match } m \text{ rs, Undecided} \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, \text{pl}_g \langle \text{rs, Undecided} \rangle$
 $\Rightarrow t$ **(is ?l \longleftrightarrow ?r)**
⟨proof⟩ **lemma** *matches-add-match-simp*:
 $\text{matches } \gamma \text{ m p} \implies \Gamma, \gamma, \text{pl}_g \langle \text{add-match } m \text{ rs, s} \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, \text{pl}_g \langle \text{rs, s} \rangle \Rightarrow t$
⟨proof⟩ **lemma** *not-matches-add-matchNot-simp*:
 $\neg \text{matches } \gamma \text{ m p} \implies \Gamma, \gamma, \text{pl}_g \langle \text{add-match (MatchNot m) rs, s} \rangle \Rightarrow t \longleftrightarrow$
 $\Gamma, \gamma, \text{pl}_g \langle \text{rs, s} \rangle \Rightarrow t$
⟨proof⟩

17.4 Goto Unfolding

private lemma *unfold-Goto-Undecided*:
assumes *chain-defined*: $\Gamma \text{ chain} = \text{Some rs and no-matching-Goto-rs}$
no-matching-Goto γ p rs
shows $\Gamma, \gamma, \text{pl}_g \langle (\text{Rule } m \text{ (Goto chain)})\# \text{rest, Undecided} \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, \text{pl}_g$
 $\langle \text{add-match } m \text{ rs @ add-match (MatchNot m) rest, Undecided} \rangle \Rightarrow t$
(is ?l \longleftrightarrow ?r)
⟨proof⟩ **theorem** *unfold-Goto*:
assumes *chain-defined*: $\Gamma \text{ chain} = \text{Some rs and no-matching-Goto-rs}$
no-matching-Goto γ p rs
shows $\Gamma, \gamma, \text{pl}_g \langle (\text{Rule } m \text{ (Goto chain)})\# \text{rest, s} \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, \text{pl}_g \langle \text{add-match}$
 $m \text{ rs @ add-match (MatchNot m) rest, s} \rangle \Rightarrow t$
⟨proof⟩

A chain that will definitely come to a direct decision

qualified fun *terminal-chain* :: 'a rule list \Rightarrow bool **where**

terminal-chain [] = False |
terminal-chain [Rule MatchAny Accept] = True |
terminal-chain [Rule MatchAny Drop] = True |
terminal-chain [Rule MatchAny Reject] = True |
terminal-chain ((Rule - (Goto -))#rs) = False |
terminal-chain ((Rule - (Call -))#rs) = False |
terminal-chain ((Rule - Return)#rs) = False |
terminal-chain ((Rule - Unknown)#rs) = False |
terminal-chain (-#rs) = *terminal-chain* rs

private lemma *terminal-chain-no-matching-Goto*: *terminal-chain* rs \Longrightarrow *no-matching-Goto*
 γ p rs
 (proof)

A terminal chain means (if the semantics are actually defined) that the chain will ultimately yield a final filtering decision, for all packets.

qualified lemma *terminal-chain* rs \Longrightarrow $\Gamma, \gamma, p \vdash_g \langle rs, \text{Undecided} \rangle \Rightarrow t \Longrightarrow \exists X. t = \text{Decision } X$

(proof) **lemma** *replace-Goto-with-Call-in-terminal-chain-Undecided*:

assumes *chain-defined*: Γ chain = Some rs **and** *terminal-chain*: *terminal-chain* rs

shows $\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ (Goto\ chain)], \text{Undecided} \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, p \vdash_g \langle [Rule\ m\ (Call\ chain)], \text{Undecided} \rangle \Rightarrow t$

(is ?l \longleftrightarrow ?r)

(proof) **theorem** *replace-Goto-with-Call-in-terminal-chain*:

assumes *chain-defined*: Γ chain = Some rs **and** *terminal-chain*: *terminal-chain* rs

shows $\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ (Goto\ chain)], s \rangle \Rightarrow t \longleftrightarrow \Gamma, \gamma, p \vdash_g \langle [Rule\ m\ (Call\ chain)], s \rangle \Rightarrow t$

(proof) **fun** *rewrite-Goto-chain-safe* :: (string \rightarrow 'a rule list) \Rightarrow 'a rule list \Rightarrow ('a rule list) option **where**

rewrite-Goto-chain-safe - [] = Some [] |

rewrite-Goto-chain-safe Γ ((Rule m (Goto chain))#rs) =

(case (Γ chain) of None \Rightarrow None

| Some rs' \Rightarrow (if

\neg *terminal-chain* rs'

then

None

else

map-option (λ rs. Rule m (Call chain) # rs)

(*rewrite-Goto-chain-safe* Γ rs)

)

) |

rewrite-Goto-chain-safe Γ (r#rs) = map-option (λ rs. r # rs) (*rewrite-Goto-chain-safe* Γ rs)

private fun *rewrite-Goto-safe-internal*

:: (string \times 'a rule list) list \Rightarrow (string \times 'a rule list) list \Rightarrow (string \times 'a rule

list) list option where
 rewrite-Goto-safe-internal - [] = Some [] |
 rewrite-Goto-safe-internal Γ ((chain-name, rs)#cs) =
 (case rewrite-Goto-chain-safe (map-of Γ) rs of
 None \Rightarrow None
 | Some rs' \Rightarrow map-option (λ rst. (chain-name, rs')#rst)
 (rewrite-Goto-safe-internal Γ cs)
)

qualified fun rewrite-Goto-safe :: (string \times 'a rule list) list \Rightarrow (string \times 'a rule list) list option where
 rewrite-Goto-safe cs = rewrite-Goto-safe-internal cs cs

qualified definition rewrite-Goto :: (string \times 'a rule list) list \Rightarrow (string \times 'a rule list) list where
 rewrite-Goto cs = the (rewrite-Goto-safe cs)

private lemma step-IH-cong: (\bigwedge s. $\Gamma, \gamma, p \vdash_g \langle rs1, s \rangle \Rightarrow t = \Gamma, \gamma, p \vdash_g \langle rs2, s \rangle \Rightarrow t$) \Rightarrow
 $\Gamma, \gamma, p \vdash_g \langle r\#rs1, s \rangle \Rightarrow t = \Gamma, \gamma, p \vdash_g \langle r\#rs2, s \rangle \Rightarrow t$
 (proof) **lemma** terminal-chain-decision:
 terminal-chain rs $\Rightarrow \Gamma, \gamma, p \vdash_g \langle rs, Undecided \rangle \Rightarrow t \Rightarrow \exists X. t = Decision X$
 (proof) **lemma** terminal-chain-Goto-decision: Γ chain = Some rs \Rightarrow terminal-chain rs \Rightarrow matches γ m p \Rightarrow
 $\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ (Goto\ chain)], s \rangle \Rightarrow t \Rightarrow \exists X. t = Decision X$
 (proof) **theorem** rewrite-Goto-chain-safe:
 rewrite-Goto-chain-safe Γ rs = Some rs' $\Rightarrow \Gamma, \gamma, p \vdash_g \langle rs', s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash_g \langle rs, s \rangle \Rightarrow t$
 (proof)

Example: The semantics are actually defined (for this example).

lemma defines $\gamma \equiv (\lambda - . True)$ and $m \equiv MatchAny$
shows ["FORWARD" \mapsto [Rule m Log, Rule m (Call "foo"), Rule m Drop],
 "foo" \mapsto [Rule m Log, Rule m (Goto "bar"), Rule m Reject],
 "bar" \mapsto [Rule m (Goto "baz"), Rule m Reject],
 "baz" \mapsto [(Rule m Accept)]],
 $\gamma, p \vdash_g \langle [Rule\ MatchAny\ (Call\ "FORWARD")], Undecided \rangle \Rightarrow (Decision\ FinalAllow)$
 (proof)

end

end

18 Negation Type DNF

```

theory Negation-Type-DNF
imports Negation-Type
begin

```

```

type-synonym 'a dnf = (('a negation-type) list) list

```

```

fun cnf-to-bool :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a negation-type list  $\Rightarrow$  bool where
  cnf-to-bool - []  $\longleftrightarrow$  True |
  cnf-to-bool f (Pos a#as)  $\longleftrightarrow$  (f a)  $\wedge$  cnf-to-bool f as |
  cnf-to-bool f (Neg a#as)  $\longleftrightarrow$  ( $\neg$  f a)  $\wedge$  cnf-to-bool f as

```

```

fun dnf-to-bool :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a dnf  $\Rightarrow$  bool where
  dnf-to-bool - []  $\longleftrightarrow$  False |
  dnf-to-bool f (as#ass)  $\longleftrightarrow$  (cnf-to-bool f as)  $\vee$  (dnf-to-bool f ass)

```

representing *True*

```

definition dnf-True :: 'a dnf where

```

```

  dnf-True  $\equiv$  [[]]

```

```

lemma dnf-True: dnf-to-bool f dnf-True
  <proof>

```

representing *False*

```

definition dnf-False :: 'a dnf where

```

```

  dnf-False  $\equiv$  []

```

```

lemma dnf-False:  $\neg$  dnf-to-bool f dnf-False
  <proof>

```

```

lemma cnf-to-bool-append: cnf-to-bool  $\gamma$  (a1 @ a2)  $\longleftrightarrow$  cnf-to-bool  $\gamma$  a1  $\wedge$  cnf-to-bool
 $\gamma$  a2
  <proof>

```

```

lemma dnf-to-bool-append: dnf-to-bool  $\gamma$  (a1 @ a2)  $\longleftrightarrow$  dnf-to-bool  $\gamma$  a1  $\vee$  dnf-to-bool
 $\gamma$  a2
  <proof>

```

```

definition dnf-and :: 'a dnf  $\Rightarrow$  'a dnf  $\Rightarrow$  'a dnf where

```

```

  dnf-and cnf1 cnf2 = [andlist1 @ andlist2. andlist1 <- cnf1, andlist2 <- cnf2]

```

```

value dnf-and ([[a,b], [c,d]]) ([[v,w], [x,y]])

```

```

lemma cnf-to-bool-set: cnf-to-bool f cnf  $\longleftrightarrow$  ( $\forall$  c  $\in$  set cnf. (case c of Pos a  $\Rightarrow$  f
a | Neg a  $\Rightarrow$   $\neg$  f a))
  <proof>

```

```

lemma dnf-to-bool-set: dnf-to-bool  $\gamma$  dnf  $\longleftrightarrow$  ( $\exists$  d  $\in$  set dnf. cnf-to-bool  $\gamma$  d)
  <proof>

```

lemma *dnf-to-bool-seteq*: $set \text{ ' } set \ d1 = set \text{ ' } set \ d2 \implies dnf\text{-to-bool} \ \gamma \ d1 \longleftrightarrow dnf\text{-to-bool} \ \gamma \ d2$
 ⟨proof⟩

lemma *dnf-and-correct*: $dnf\text{-to-bool} \ \gamma \ (dnf\text{-and} \ d1 \ d2) \longleftrightarrow dnf\text{-to-bool} \ \gamma \ d1 \wedge dnf\text{-to-bool} \ \gamma \ d2$
 ⟨proof⟩

lemma *dnf-and-symmetric*: $dnf\text{-to-bool} \ \gamma \ (dnf\text{-and} \ d1 \ d2) \longleftrightarrow dnf\text{-to-bool} \ \gamma \ (dnf\text{-and} \ d2 \ d1)$
 ⟨proof⟩

18.0.1 inverting a DNF

Example

lemma $(\neg ((a1 \wedge a2) \vee b \vee c)) = ((\neg a1 \wedge \neg b \wedge \neg c) \vee (\neg a2 \wedge \neg b \wedge \neg c))$
 ⟨proof⟩

lemma $(\neg ((a1 \wedge a2) \vee (b1 \wedge b2) \vee c)) = ((\neg a1 \wedge \neg b1 \wedge \neg c) \vee (\neg a2 \wedge \neg b1 \wedge \neg c) \vee (\neg a1 \wedge \neg b2 \wedge \neg c) \vee (\neg a2 \wedge \neg b2 \wedge \neg c))$ ⟨proof⟩

fun *listprepend* :: 'a list ⇒ 'a list list ⇒ 'a list list **where**
listprepend [] ns = [] |
listprepend (a#as) ns = (map (λxs. a#xs) ns) @ (*listprepend* as ns)

lemma *listprepend* [a,b] [as, bs] = [a#as, a#bs, b#as, b#bs] ⟨proof⟩

lemma *map-a-and*: $dnf\text{-to-bool} \ \gamma \ (map \ ((\#) \ a) \ ds) \longleftrightarrow dnf\text{-to-bool} \ \gamma \ [[a]] \wedge dnf\text{-to-bool} \ \gamma \ ds$
 ⟨proof⟩

this is how *listprepend* works:

lemma $\neg dnf\text{-to-bool} \ \gamma \ (listprepend \ [] \ ds)$ ⟨proof⟩

lemma $dnf\text{-to-bool} \ \gamma \ (listprepend \ [a] \ ds) \longleftrightarrow dnf\text{-to-bool} \ \gamma \ [[a]] \wedge dnf\text{-to-bool} \ \gamma \ ds$ ⟨proof⟩

lemma $dnf\text{-to-bool} \ \gamma \ (listprepend \ [a, b] \ ds) \longleftrightarrow (dnf\text{-to-bool} \ \gamma \ [[a]] \wedge dnf\text{-to-bool} \ \gamma \ ds) \vee (dnf\text{-to-bool} \ \gamma \ [[b]] \wedge dnf\text{-to-bool} \ \gamma \ ds)$
 ⟨proof⟩

We use \exists to model the big \vee operation

lemma *listprepend-correct*: $dnf\text{-to-bool} \ \gamma \ (listprepend \ as \ ds) \longleftrightarrow (\exists a \in set \ as. dnf\text{-to-bool} \ \gamma \ [[a]] \wedge dnf\text{-to-bool} \ \gamma \ ds)$
 ⟨proof⟩

lemma *listprepend-correct'*: $dnf\text{-to-bool} \ \gamma \ (listprepend \ as \ ds) \longleftrightarrow (dnf\text{-to-bool} \ \gamma \ (map \ (\lambda a. [a]) \ as) \wedge dnf\text{-to-bool} \ \gamma \ ds)$
 ⟨proof⟩

lemma *cnf-invert-singelton*: $cnf\text{-to-bool} \ \gamma \ [invert \ a] \longleftrightarrow \neg cnf\text{-to-bool} \ \gamma \ [a]$
 ⟨proof⟩

lemma *cnf-singleton-false*: $(\exists a' \in \text{set } as. \neg \text{cnf-to-bool } \gamma [a']) \longleftrightarrow \neg \text{cnf-to-bool } \gamma as$
 <proof>

fun *dnf-not* :: 'a dnf \Rightarrow 'a dnf **where**
dnf-not [] = [] |
dnf-not (ns#nss) = listprepend (map invert ns) (dnf-not nss)

lemma *dnf-not*: $\text{dnf-to-bool } \gamma (\text{dnf-not } d) \longleftrightarrow \neg \text{dnf-to-bool } \gamma d$
 <proof>

18.0.2 Optimizing

definition *optimize-dfn* :: 'a dnf \Rightarrow 'a dnf **where**
optimize-dfn dnf = map remdups (remdups dnf)

lemma *dnf-to-bool f* (*optimize-dfn* dnf) = *dnf-to-bool f* dnf
 <proof>

end

theory *Matching-Embeddings*

imports *Semantics-Ternary/Matching-Ternary Matching Semantics-Ternary/Unknown-Match-Tacs*
begin

19 Boolean Matching vs. Ternary Matching

term *Semantics.matches*

term *Matching-Ternary.matches*

The two matching semantics are related. However, due to the ternary logic, we cannot directly translate one to the other. The problem are *MatchNot* expressions which evaluate to *TernaryUnknown* because *MatchNot TernaryUnknown* and *TernaryUnknown* are semantically equal!

lemma $\exists m \beta \alpha a. \text{Matching-Ternary.matches } (\beta, \alpha) m a p \neq$
Semantics.matches ($\lambda atm p. \text{case } \beta atm p \text{ of TernaryTrue} \Rightarrow \text{True} \mid \text{TernaryFalse} \Rightarrow \text{False} \mid \text{TernaryUnknown} \Rightarrow \alpha a p$) *m p*
 <proof>

the *the* in the next definition is always defined

lemma $\forall m \in \{m. \text{approx } m p \neq \text{TernaryUnknown}\}. \text{ternary-to-bool } (\text{approx } m p) \neq \text{None}$
 <proof>

The Boolean and the ternary matcher agree (where the ternary matcher is defined)

definition *matcher-agree-on-exact-matches* :: ('a, 'p) matcher \Rightarrow ('a \Rightarrow 'p \Rightarrow ternaryvalue) \Rightarrow bool **where**

matcher-agree-on-exact-matches exact approx $\equiv \forall p m. \text{approx } m p \neq \text{TernaryUnknown} \longrightarrow \text{exact } m p = \text{the } (\text{ternary-to-bool } (\text{approx } m p))$

We say the Boolean and ternary matchers agree iff they return the same result or the ternary matcher returns *TernaryUnknown*.

lemma *matcher-agree-on-exact-matches exact approx* $\longleftrightarrow (\forall p m. \text{exact } m p = \text{the } (\text{ternary-to-bool } (\text{approx } m p)) \vee \text{approx } m p = \text{TernaryUnknown})$

<proof>

lemma *matcher-agree-on-exact-matches-alt:*

matcher-agree-on-exact-matches exact approx $\longleftrightarrow (\forall p m. \text{approx } m p \neq \text{TernaryUnknown} \longrightarrow \text{bool-to-ternary } (\text{exact } m p) = \text{approx } m p)$

<proof>

lemma *eval-ternary-Not-TrueD:* *eval-ternary-Not* $m = \text{TernaryTrue} \implies m = \text{TernaryFalse}$

<proof>

lemma *matches-comply-exact: ternary-ternary-eval* $(\text{map-match-tac } \beta p m) \neq \text{TernaryUnknown} \implies$

matcher-agree-on-exact-matches $\gamma \beta \implies$

Semantics.matches $\gamma m p = \text{Matching-Ternary.matches } (\beta, \alpha) m a p$

<proof>

lemma *matcher-agree-on-exact-matches-gammaE:*

matcher-agree-on-exact-matches $\gamma \beta \implies \beta X p = \text{TernaryTrue} \implies \gamma X p$

<proof>

lemma *in-doubt-allow-allows-Accept:* $a = \text{Accept} \implies \text{matcher-agree-on-exact-matches } \gamma \beta \implies$

Semantics.matches $\gamma m p \implies \text{Matching-Ternary.matches } (\beta, \text{in-doubt-allow})$

$m a p$

<proof>

lemma *not-exact-match-in-doubt-allow-approx-match:* *matcher-agree-on-exact-matches* $\gamma \beta \implies a = \text{Accept} \vee a = \text{Reject} \vee a = \text{Drop} \implies$

$\neg \text{Semantics.matches } \gamma m p \implies$

$(a = \text{Accept} \wedge \text{Matching-Ternary.matches } (\beta, \text{in-doubt-allow}) m a p) \vee \neg \text{Matching-Ternary.matches } (\beta, \text{in-doubt-allow}) m a p$

<proof>

lemma *in-doubt-deny-denies-DropReject:* $a = \text{Drop} \vee a = \text{Reject} \implies \text{matcher-agree-on-exact-matches}$

$\gamma \beta \implies$
 $\text{Semantics.matches } \gamma \ m \ p \implies \text{Matching-Ternary.matches } (\beta, \text{in-doubt-deny})$
 $m \ a \ p$
 $\langle \text{proof} \rangle$

lemma *not-exact-match-in-doubt-deny-approx-match: matcher-agree-on-exact-matches*
 $\gamma \beta \implies a = \text{Accept} \vee a = \text{Reject} \vee a = \text{Drop} \implies$
 $\neg \text{Semantics.matches } \gamma \ m \ p \implies$
 $((a = \text{Drop} \vee a = \text{Reject}) \wedge \text{Matching-Ternary.matches } (\beta, \text{in-doubt-deny}) \ m \ a$
 $p) \vee \neg \text{Matching-Ternary.matches } (\beta, \text{in-doubt-deny}) \ m \ a \ p$
 $\langle \text{proof} \rangle$

The ternary primitive matcher can return exactly the result of the Boolean primitive matcher

definition $\beta_{\text{magic}} :: ('a, 'p) \text{ matcher} \Rightarrow ('a \Rightarrow 'p \Rightarrow \text{ternaryvalue})$ **where**
 $\beta_{\text{magic}} \ \gamma \equiv (\lambda \ a \ p. \text{ if } \gamma \ a \ p \text{ then TernaryTrue else TernaryFalse})$

lemma *matcher-agree-on-exact-matches* $\gamma \ (\beta_{\text{magic}} \ \gamma)$
 $\langle \text{proof} \rangle$

lemma *β_{magic} -not-Unknown: ternary-ternary-eval (map-match-tac ($\beta_{\text{magic}} \ \gamma$) p*
 $m) \neq \text{TernaryUnknown}$
 $\langle \text{proof} \rangle$

lemma *β_{magic} -matching: Matching-Ternary.matches (($\beta_{\text{magic}} \ \gamma$), α) m a p \longleftrightarrow*
 $\text{Semantics.matches } \gamma \ m \ p$
 $\langle \text{proof} \rangle$

end
theory *Fixed-Action*
imports *Semantics-Ternary*
begin

20 Fixed Action

If firewall rules have the same action, we can focus on the matching only.

Applying a rule once or several times makes no difference.

lemma *approximating-bigstep-fun-prepend-replicate:*
 $n > 0 \implies \text{approximating-bigstep-fun } \gamma \ p \ (r\#rs) \ \text{Undecided} = \text{approximat-}$
 $\text{ing-bigstep-fun } \gamma \ p \ ((\text{replicate } n \ r)\@rs) \ \text{Undecided}$
 $\langle \text{proof} \rangle$

utility lemmas

context
begin

private lemma *fixedaction-Log: approximating-bigstep-fun* γ p (map (λm . Rule m Log) ms) Undecided = Undecided
 ⟨proof⟩ **lemma** *fixedaction-Empty: approximating-bigstep-fun* γ p (map (λm . Rule m Empty) ms) Undecided = Undecided
 ⟨proof⟩ **lemma** *helperX1-Log: matches* γ m' Log $p \implies$
 approximating-bigstep-fun γ p (map ((λm . Rule m Log) \circ MatchAnd m')
 $m2' @ rs2$) Undecided =
 approximating-bigstep-fun γ p $rs2$ Undecided
 ⟨proof⟩ **lemma** *helperX1-Empty: matches* γ m' Empty $p \implies$
 approximating-bigstep-fun γ p (map ((λm . Rule m Empty) \circ MatchAnd m')
 $m2' @ rs2$) Undecided =
 approximating-bigstep-fun γ p $rs2$ Undecided
 ⟨proof⟩ **lemma** *helperX3: matches* γ m' a $p \implies$
 approximating-bigstep-fun γ p (map ((λm . Rule m a) \circ MatchAnd m') $m2'$
 $@ rs2$) Undecided =
 approximating-bigstep-fun γ p (map (λm . Rule m a) $m2' @ rs2$) Undecided
 ⟨proof⟩

lemmas *fixed-action-simps = fixedaction-Log fixedaction-Empty helperX1-Log
 helperX1-Empty helperX3*
end

lemma *fixedaction-swap:*

approximating-bigstep-fun γ p (map (λm . Rule m a) ($m1@m2$)) s = approxi-
 mating-bigstep-fun γ p (map (λm . Rule m a) ($m2@m1$)) s
 ⟨proof⟩

corollary *fixedaction-reorder: approximating-bigstep-fun* γ p (map (λm . Rule m a)
 ($m1 @ m2 @ m3$)) s = approximating-bigstep-fun γ p (map (λm . Rule m a) ($m2$
 $@ m1 @ m3$)) s
 ⟨proof⟩

If the actions are equal, the *set* (position and replication independent) of the match expressions can be considered.

lemma *approximating-bigstep-fun-fixaction-matchseteq: set* $m1 = set$ $m2 \implies$
 approximating-bigstep-fun γ p (map (λm . Rule m a) $m1$) s =
 approximating-bigstep-fun γ p (map (λm . Rule m a) $m2$) s
 ⟨proof⟩

20.1 match-list

Reducing the firewall semantics to short-circuit matching evaluation

fun *match-list* :: ('a, 'packet) match-tac \Rightarrow 'a match-expr list \Rightarrow action \Rightarrow 'packet
 \Rightarrow bool **where**
 match-list γ [] a p = False |
 match-list γ ($m\#ms$) a p = (if matches γ m a p then True else match-list γ ms
 a p)

lemma *match-list-matches*: $match\text{-}list\ \gamma\ ms\ a\ p \longleftrightarrow (\exists m \in set\ ms.\ matches\ \gamma\ m\ a\ p)$
 ⟨proof⟩

lemma *match-list-True*: $match\text{-}list\ \gamma\ ms\ a\ p \implies approximating\text{-}bigstep\text{-}fun\ \gamma\ p$
 $(map\ (\lambda m.\ Rule\ m\ a)\ ms)\ Undecided = (case\ a\ of\ Accept \implies Decision\ FinalAllow$
 | $Drop \implies Decision\ FinalDeny$
 | $Reject \implies Decision\ FinalDeny$
 | $Log \implies Undecided$
 | $Empty \implies Undecided$
 — unhandled cases
)

⟨proof⟩

lemma *match-list-False*: $\neg match\text{-}list\ \gamma\ ms\ a\ p \implies approximating\text{-}bigstep\text{-}fun\ \gamma\ p$
 $(map\ (\lambda m.\ Rule\ m\ a)\ ms)\ Undecided = Undecided$
 ⟨proof⟩

The key idea behind *match-list*: Reducing semantics to match list

lemma *match-list-semantics*: $match\text{-}list\ \gamma\ ms1\ a\ p \longleftrightarrow match\text{-}list\ \gamma\ ms2\ a\ p \implies$
 $approximating\text{-}bigstep\text{-}fun\ \gamma\ p\ (map\ (\lambda m.\ Rule\ m\ a)\ ms1)\ s = approximating\text{-}bigstep\text{-}fun\ \gamma\ p\ (map\ (\lambda m.\ Rule\ m\ a)\ ms2)\ s$
 ⟨proof⟩

We can exploit de-morgan to get a disjunction in the match expression!

fun *match-list-to-match-expr* :: 'a match-expr list \implies 'a match-expr **where**
match-list-to-match-expr [] = MatchNot MatchAny |
match-list-to-match-expr (m#ms) = MatchOr m (*match-list-to-match-expr* ms)

match-list-to-match-expr constructs a unwieldy 'a match-expr from a list. The semantics of the resulting match expression is the disjunction of the elements of the list. This is handy because the normal match expressions do not directly support disjunction. Use this function with care because the resulting match expression is very ugly!

lemma *match-list-to-match-expr-disjunction*: $match\text{-}list\ \gamma\ ms\ a\ p \longleftrightarrow matches\ \gamma\ (match\text{-}list\text{-}to\text{-}match\text{-}expr\ ms)\ a\ p$
 ⟨proof⟩

lemma *match-list-singleton*: $match\text{-}list\ \gamma\ [m]\ a\ p \longleftrightarrow matches\ \gamma\ m\ a\ p$ ⟨proof⟩

lemma *match-list-append*: $match\text{-}list\ \gamma\ (m1@m2)\ a\ p \longleftrightarrow (\neg match\text{-}list\ \gamma\ m1\ a\ p \longrightarrow match\text{-}list\ \gamma\ m2\ a\ p)$
 ⟨proof⟩

lemma *match-list-helper1*: $\neg matches\ \gamma\ m2\ a\ p \implies match\text{-}list\ \gamma\ (map\ (\lambda x.\ MatchAnd\ x\ m2)\ m1')\ a\ p \implies False$
 ⟨proof⟩

lemma *match-list-helper2*: $\neg matches\ \gamma\ m\ a\ p \implies \neg match\text{-}list\ \gamma\ (map\ (MatchAnd\ m)\ m2')\ a\ p$

<proof>

lemma *match-list-helper3*: $\text{matches } \gamma \ m \ a \ p \implies \text{match-list } \gamma \ m2' \ a \ p \implies \text{match-list } \gamma \ (\text{map } (\text{MatchAnd } m) \ m2') \ a \ p$

<proof>

lemma *match-list-helper4*: $\neg \text{match-list } \gamma \ m2' \ a \ p \implies \neg \text{match-list } \gamma \ (\text{map } (\text{MatchAnd } aa) \ m2') \ a \ p$

<proof>

lemma *match-list-helper5*: $\neg \text{match-list } \gamma \ m2' \ a \ p \implies \neg \text{match-list } \gamma \ (\text{concat } (\text{map } (\lambda x. \text{map } (\text{MatchAnd } x) \ m2') \ m1')) \ a \ p$

<proof>

lemma *match-list-helper6*: $\neg \text{match-list } \gamma \ m1' \ a \ p \implies \neg \text{match-list } \gamma \ (\text{concat } (\text{map } (\lambda x. \text{map } (\text{MatchAnd } x) \ m2') \ m1')) \ a \ p$

<proof>

lemmas *match-list-helper* = *match-list-helper1 match-list-helper2 match-list-helper3 match-list-helper4 match-list-helper5 match-list-helper6*

hide-fact *match-list-helper1 match-list-helper2 match-list-helper3 match-list-helper4 match-list-helper5 match-list-helper6*

lemma *match-list-map-And1*: $\text{matches } \gamma \ m1 \ a \ p = \text{match-list } \gamma \ m1' \ a \ p \implies \text{matches } \gamma \ (\text{MatchAnd } m1 \ m2) \ a \ p \iff \text{match-list } \gamma \ (\text{map } (\lambda x. \text{MatchAnd } x \ m2) \ m1') \ a \ p$

<proof>

lemma *matches-list-And-concat*: $\text{matches } \gamma \ m1 \ a \ p = \text{match-list } \gamma \ m1' \ a \ p \implies \text{matches } \gamma \ m2 \ a \ p = \text{match-list } \gamma \ m2' \ a \ p \implies$

$\text{matches } \gamma \ (\text{MatchAnd } m1 \ m2) \ a \ p \iff \text{match-list } \gamma \ [\text{MatchAnd } x \ y. \ x \ <- \ m1', \ y \ <- \ m2'] \ a \ p$

<proof>

lemma *match-list-concat*: $\text{match-list } \gamma \ (\text{concat } lss) \ a \ p \iff (\exists ls \in \text{set } lss. \text{match-list } \gamma \ ls \ a \ p)$

<proof>

lemma *fixedaction-wf-ruleset*: $\text{wf-ruleset } \gamma \ p \ (\text{map } (\lambda m. \text{Rule } m \ a) \ ms) \iff$

$\neg \text{match-list } \gamma \ ms \ a \ p \vee \neg (\exists \text{chain}. a = \text{Call chain}) \wedge a \neq \text{Return} \wedge \neg (\exists \text{chain}. a = \text{Goto chain}) \wedge a \neq \text{Unknown}$

<proof>

lemma *wf-ruleset-singleton*: $\text{wf-ruleset } \gamma \ p \ [\text{Rule } m \ a] \iff \neg \text{matches } \gamma \ m \ a \ p \vee \neg (\exists \text{chain}. a = \text{Call chain}) \wedge a \neq \text{Return} \wedge \neg (\exists \text{chain}. a = \text{Goto chain}) \wedge a \neq \text{Unknown}$

<proof>

end

```

theory Normalized-Matches
imports Fixed-Action
begin

```

21 Normalized (DNF) matches

simplify a match expression. The output is a list of match expressions, the semantics is \vee of the list elements.

```

fun normalize-match :: 'a match-expr  $\Rightarrow$  'a match-expr list where
  normalize-match (MatchAny) = [MatchAny] |
  normalize-match (Match m) = [Match m] |
  normalize-match (MatchAnd m1 m2) = [MatchAnd x y. x <- normalize-match
m1, y <- normalize-match m2] |
  normalize-match (MatchNot (MatchAnd m1 m2)) = normalize-match (MatchNot
m1) @ normalize-match (MatchNot m2) |
  normalize-match (MatchNot (MatchNot m)) = normalize-match m |
  normalize-match (MatchNot (MatchAny)) = [] |
  normalize-match (MatchNot (Match m)) = [MatchNot (Match m)]

```

lemma *normalize-match-not-matcheq-matchNone*: $\forall m' \in \text{set } (\text{normalize-match } m).$
 $\neg \text{matcheq-matchNone } m'$
<proof>

lemma *normalize-match-empty-iff-matcheq-matchNone*: $\text{normalize-match } m = []$
 $\iff \text{matcheq-matchNone } m$
<proof>

lemma *match-list-normalize-match*: $\text{match-list } \gamma [m] a p \iff \text{match-list } \gamma (\text{normalize-match } m) a p$
<proof>

thm *match-list-normalize-match[simplified match-list-singleton]*

theorem *normalize-match-correct*: $\text{approximating-bigstep-fun } \gamma p (\text{map } (\lambda m. \text{Rule } m a) (\text{normalize-match } m)) s = \text{approximating-bigstep-fun } \gamma p [\text{Rule } m a] s$
<proof>

lemma *normalize-match-empty*: $\text{normalize-match } m = [] \implies \neg \text{matches } \gamma m a p$
<proof>

lemma *matches-to-match-list-normalize*: $\text{matches } \gamma m a p = \text{match-list } \gamma (\text{normalize-match } m) a p$
<proof>

lemma *wf-ruleset-normalize-match*: $wf\text{-ruleset } \gamma p [(Rule\ m\ a)] \implies wf\text{-ruleset } \gamma p (map\ (\lambda m. Rule\ m\ a)\ (normalize\text{-match}\ m))$
 $\langle proof \rangle$

lemma *normalize-match-wf-ruleset*: $wf\text{-ruleset } \gamma p (map\ (\lambda m. Rule\ m\ a)\ (normalize\text{-match}\ m)) \implies wf\text{-ruleset } \gamma p [Rule\ m\ a]$
 $\langle proof \rangle$

lemma *good-ruleset-normalize-match*: $good\text{-ruleset } [(Rule\ m\ a)] \implies good\text{-ruleset } (map\ (\lambda m. Rule\ m\ a)\ (normalize\text{-match}\ m))$
 $\langle proof \rangle$

22 Normalizing rules instead of only match expressions

fun *normalize-rules* :: ('a match-expr \Rightarrow 'a match-expr list) \Rightarrow 'a rule list \Rightarrow 'a rule list **where**
 $normalize\text{-rules } - [] = [] \mid$
 $normalize\text{-rules } f ((Rule\ m\ a)\#rs) = (map\ (\lambda m. Rule\ m\ a)\ (f\ m))\@(normalize\text{-rules } f\ rs)$

lemma *normalize-rules-singleton*: $normalize\text{-rules } f [Rule\ m\ a] = map\ (\lambda m. Rule\ m\ a)\ (f\ m)$ $\langle proof \rangle$

lemma *normalize-rules-fst*: $(normalize\text{-rules } f\ (r\ \#\ rs)) = (normalize\text{-rules } f\ [r])\@(normalize\text{-rules } f\ rs)$
 $\langle proof \rangle$

lemma *normalize-rules-concat-map*:
 $normalize\text{-rules } f\ rs = concat\ (map\ (\lambda r. map\ (\lambda m. Rule\ m\ (get\text{-action}\ r))\ (f\ (get\text{-match}\ r)))\ rs)$
 $\langle proof \rangle$

lemma *good-ruleset-normalize-rules*: $good\text{-ruleset } rs \implies good\text{-ruleset } (normalize\text{-rules } f\ rs)$
 $\langle proof \rangle$

lemma *simple-ruleset-normalize-rules*: $simple\text{-ruleset } rs \implies simple\text{-ruleset } (normalize\text{-rules } f\ rs)$
 $\langle proof \rangle$

lemma *normalize-rules-match-list-semantic-3*:

assumes $\forall m\ a. P\ m \longrightarrow match\text{-list } \gamma\ (f\ m)\ a\ p = matches\ \gamma\ m\ a\ p$

and *simple-ruleset* rs

and $P: \forall r \in set\ rs. P\ (get\text{-match}\ r)$

shows *approximating-bigstep-fun* γ p (*normalize-rules* f rs) s = *approximating-bigstep-fun* γ p rs s
 ⟨*proof*⟩

corollary *normalize-rules-match-list-semantic*:
 ($\forall m a. \text{match-list } \gamma (f m) a p = \text{matches } \gamma m a p$) \implies *simple-ruleset* $rs \implies$
approximating-bigstep-fun γ p (*normalize-rules* f rs) s = *approximating-bigstep-fun*
 γ p rs s
 ⟨*proof*⟩

lemma *in-normalized-matches*: $ls \in \text{set } (\text{normalize-match } m) \wedge \text{matches } \gamma ls a p$
 $\implies \text{matches } \gamma m a p$
 ⟨*proof*⟩

applying a function (with a prerequisite Q) to all rules

lemma *normalize-rules-property*:
assumes $\forall r \in \text{set } rs. P (\text{get-match } r)$
and $\forall m. P m \longrightarrow (\forall m' \in \text{set } (f m). Q m')$
shows $\forall r \in \text{set } (\text{normalize-rules } f rs). Q (\text{get-match } r)$
 ⟨*proof*⟩

If a function f preserves some property of the match expressions, then this property is preserved when applying *normalize-rules*

lemma *normalize-rules-preserves*: **assumes** $\forall r \in \text{set } rs. P (\text{get-match } r)$
and $\forall m. P m \longrightarrow (\forall m' \in \text{set } (f m). P m')$
shows $\forall r \in \text{set } (\text{normalize-rules } f rs). P (\text{get-match } r)$
 ⟨*proof*⟩

fun *normalize-rules-dnf* :: 'a rule list \Rightarrow 'a rule list **where**
normalize-rules-dnf [] = [] |
normalize-rules-dnf ((Rule m a)# rs) = (map ($\lambda m. \text{Rule } m a$) (*normalize-match*
 m))@(*normalize-rules-dnf* rs)

lemma *normalize-rules-dnf-append*: *normalize-rules-dnf* ($rs1 @ rs2$) = *normalize-rules-dnf*
 $rs1 @ \text{normalize-rules-dnf } rs2$
 ⟨*proof*⟩

lemma *normalize-rules-dnf-def2*: *normalize-rules-dnf* = *normalize-rules normalize-match*
 ⟨*proof*⟩

lemma *wf-ruleset-normalize-rules-dnf*: *wf-ruleset* γ p $rs \implies \text{wf-ruleset } \gamma$ p (*normalize-rules-dnf*
 rs)
 ⟨*proof*⟩

lemma *good-ruleset-normalize-rules-dnf*: *good-ruleset* $rs \implies \text{good-ruleset } (\text{normalize-rules-dnf}$
 $rs)$
 ⟨*proof*⟩

lemma *simple-ruleset-normalize-rules-dnf*: *simple-ruleset* *rs* \implies *simple-ruleset* (*normalize-rules-dnf* *rs*)
 ⟨*proof*⟩

lemma *simple-ruleset* *rs* \implies
approximating-bigstep-fun γ *p* (*normalize-rules-dnf* *rs*) *s* = *approximating-bigstep-fun*
 γ *p* *rs* *s*
 ⟨*proof*⟩

lemma *normalize-rules-dnf-correct*: *wf-ruleset* γ *p* *rs* \implies
approximating-bigstep-fun γ *p* (*normalize-rules-dnf* *rs*) *s* = *approximating-bigstep-fun*
 γ *p* *rs* *s*
 ⟨*proof*⟩

fun *normalized-nnf-match* :: 'a *match-expr* \Rightarrow *bool* **where**
normalized-nnf-match *MatchAny* = *True* |
normalized-nnf-match (*Match* -) = *True* |
normalized-nnf-match (*MatchNot* (*Match* -)) = *True* |
normalized-nnf-match (*MatchAnd* *m1* *m2*) = ((*normalized-nnf-match* *m1*) \wedge
(*normalized-nnf-match* *m2*)) |
normalized-nnf-match - = *False*

Essentially, *normalized-nnf-match* checks for a negation normal form: Only AND is at toplevel, negation only occurs in front of literals. Since 'a *match-expr* does not support OR, the result is in conjunction normal form. Applying *normalize-match*, the result is a list. Essentially, this is the disjunctive normal form.

lemma *normalize-match-already-normalized*: *normalized-nnf-match* *m* \implies *normalize-match* *m* = [*m*]
 ⟨*proof*⟩

lemma *normalized-nnf-match-normalize-match*: \forall *m'* \in *set* (*normalize-match* *m*).
normalized-nnf-match *m'*
 ⟨*proof*⟩

lemma *normalized-nnf-match-MatchNot-D*: *normalized-nnf-match* (*MatchNot* *m*)
 \implies *normalized-nnf-match* *m*
 ⟨*proof*⟩

Example

lemma *normalize-match* (*MatchNot* (*MatchAnd* (*Match* *ip-src*) (*Match* *tcp*))) =
 [*MatchNot* (*Match* *ip-src*), *MatchNot* (*Match* *tcp*)] ⟨*proof*⟩

22.1 Functions which preserve *normalized-nnf-match*

lemma *optimize-matches-option-normalized-nnf-match*: $(\bigwedge r. r \in \text{set } rs \implies \text{normalized-nnf-match } (\text{get-match } r)) \implies$
 $(\bigwedge m m'. \text{normalized-nnf-match } m \implies f m = \text{Some } m' \implies \text{normalized-nnf-match } m')$
 $\forall r \in \text{set } (\text{optimize-matches-option } f rs). \text{normalized-nnf-match } (\text{get-match } r)$
 $\langle \text{proof} \rangle$

lemma *optimize-matches-normalized-nnf-match*: $\llbracket \forall r \in \text{set } rs. \text{normalized-nnf-match } (\text{get-match } r); \forall m. \text{normalized-nnf-match } m \longrightarrow \text{normalized-nnf-match } (f m) \rrbracket \implies$
 $\forall r \in \text{set } (\text{optimize-matches } f rs). \text{normalized-nnf-match } (\text{get-match } r)$
 $\langle \text{proof} \rangle$

lemma *normalize-rules-dnf-normalized-nnf-match*: $\forall x \in \text{set } (\text{normalize-rules-dnf } rs). \text{normalized-nnf-match } (\text{get-match } x)$
 $\langle \text{proof} \rangle$

end

theory *Negation-Type-Matching*

imports *../Common/Negation-Type-Matching-Ternary ../Datatype-Selectors Normalized-Matches*

begin

23 Negation Type Matching

Transform a '*a negation-type list*' to a '*a match-expr*' via conjunction.

fun *alist-and* :: '*a negation-type list* \Rightarrow '*a match-expr*' **where**
alist-and [] = *MatchAny* |
alist-and ((*Pos e*)#*es*) = *MatchAnd* (*Match e*) (*alist-and es*) |
alist-and ((*Neg e*)#*es*) = *MatchAnd* (*MatchNot* (*Match e*)) (*alist-and es*)

lemma *normalized-nnf-match-alist-and*: *normalized-nnf-match* (*alist-and as*)
 $\langle \text{proof} \rangle$

lemma *alist-and-append*: *matches* γ (*alist-and* (*l1* @ *l2*)) *a p* \longleftrightarrow *matches* γ
(*MatchAnd* (*alist-and l1*) (*alist-and l2*)) *a p*
 $\langle \text{proof} \rangle$

This version of *alist-and* avoids the trailing *MatchAny*. Only intended for code.

fun *alist-and'* :: '*a negation-type list* \Rightarrow '*a match-expr*' **where**
alist-and' [] = *MatchAny* |
alist-and' [*Pos e*] = *Match e* |
alist-and' [*Neg e*] = *MatchNot* (*Match e*)

$alist\text{-}and' ((Pos\ e)\#es) = MatchAnd (Match\ e) (alist\text{-}and'\ es) \mid$
 $alist\text{-}and' ((Neg\ e)\#es) = MatchAnd (MatchNot (Match\ e)) (alist\text{-}and'\ es)$

lemma *alist-and'*: $matches\ \gamma,\ \alpha\ (alist\text{-}and'\ as) = matches\ \gamma,\ \alpha\ (alist\text{-}and\ as)$
 <proof>

lemma *normalized-nnf-match-alist-and'*: $normalized\text{-}nnf\text{-}match\ (alist\text{-}and'\ as)$
 <proof>

lemma *matches-alist-and-alist-and'*:
 $matches\ \gamma\ (alist\text{-}and'\ ls)\ a\ p \longleftrightarrow matches\ \gamma\ (alist\text{-}and\ ls)\ a\ p$
 <proof>

lemma *alist-and'-append*: $matches\ \gamma\ (alist\text{-}and'\ (l1\ @\ l2))\ a\ p \longleftrightarrow matches\ \gamma\ (MatchAnd\ (alist\text{-}and'\ l1)\ (alist\text{-}and'\ l2))\ a\ p$
 <proof>

lemma *alist-and-NegPos-map-getNeg-getPos-matches*:
 $(\forall m \in set\ (getNeg\ spts). matches\ \gamma\ (MatchNot\ (Match\ (C\ m)))\ a\ p) \wedge$
 $(\forall m \in set\ (getPos\ spts). matches\ \gamma\ (Match\ (C\ m))\ a\ p)$
 \longleftrightarrow
 $matches\ \gamma\ (alist\text{-}and\ (NegPos\text{-}map\ C\ spts))\ a\ p$
 <proof>

fun *negation-type-to-match-expr-f* :: $'a \Rightarrow 'b \Rightarrow 'a\ negation\text{-}type \Rightarrow 'b\ match\text{-}expr$
where

$negation\text{-}type\text{-}to\text{-}match\text{-}expr\text{-}f\ f\ (Pos\ a) = Match\ (f\ a) \mid$
 $negation\text{-}type\text{-}to\text{-}match\text{-}expr\text{-}f\ f\ (Neg\ a) = MatchNot\ (Match\ (f\ a))$

lemma *alist-and-negation-type-to-match-expr-f-matches*:
 $matches\ \gamma\ (alist\text{-}and\ (NegPos\text{-}map\ C\ spts))\ a\ p \longleftrightarrow$
 $(\forall m \in set\ spts. matches\ \gamma\ (negation\text{-}type\text{-}to\text{-}match\text{-}expr\text{-}f\ C\ m)\ a\ p)$
 <proof>

definition *negation-type-to-match-expr* :: $'a\ negation\text{-}type \Rightarrow 'a\ match\text{-}expr$ **where**
 $negation\text{-}type\text{-}to\text{-}match\text{-}expr\ m \equiv negation\text{-}type\text{-}to\text{-}match\text{-}expr\text{-}f\ id\ m$

lemma *negation-type-to-match-expr-simps*:
 $negation\text{-}type\text{-}to\text{-}match\text{-}expr\ (Pos\ e) = (Match\ e)$
 $negation\text{-}type\text{-}to\text{-}match\text{-}expr\ (Neg\ e) = (MatchNot\ (Match\ e))$
 <proof>

lemma *alist-and-negation-type-to-match-expr*: $alist\text{-}and\ (n\#es) = MatchAnd\ (negation\text{-}type\text{-}to\text{-}match\text{-}expr\ n)\ (alist\text{-}and\ es)$
 <proof>

fun *to-negation-type-nnf* :: $'a\ match\text{-}expr \Rightarrow 'a\ negation\text{-}type\ list$ **where**

```

to-negation-type-nnf MatchAny = [] |
to-negation-type-nnf (Match a) = [Pos a] |
to-negation-type-nnf (MatchNot (Match a)) = [Neg a] |
to-negation-type-nnf (MatchAnd a b) = (to-negation-type-nnf a) @ (to-negation-type-nnf
b) |
to-negation-type-nnf - = undefined

```

lemma *normalized-nnf-match* $m \implies \text{matches } \gamma \text{ (alist-and (to-negation-type-nnf } m)) \text{ a } p = \text{matches } \gamma \text{ m a } p$
 ⟨proof⟩

Isolating the matching semantics

```

fun nt-match-list :: ('a, 'packet) match-tac  $\Rightarrow$  action  $\Rightarrow$  'packet  $\Rightarrow$  'a negation-type
list  $\Rightarrow$  bool where
  nt-match-list - - - [] = True |
  nt-match-list  $\gamma$  a p ((Pos x)#xs)  $\longleftrightarrow$  matches  $\gamma$  (Match x) a p  $\wedge$  nt-match-list
 $\gamma$  a p xs |
  nt-match-list  $\gamma$  a p ((Neg x)#xs)  $\longleftrightarrow$  matches  $\gamma$  (MatchNot (Match x)) a p  $\wedge$ 
nt-match-list  $\gamma$  a p xs

```

lemma *nt-match-list-matches*: $\text{nt-match-list } \gamma \text{ a } p \text{ l} \longleftrightarrow \text{matches } \gamma \text{ (alist-and l) a } p$
 ⟨proof⟩

lemma *nt-match-list-simp*: $\text{nt-match-list } \gamma \text{ a } p \text{ ms} \longleftrightarrow$
 $(\forall m \in \text{set (getPos ms)}. \text{matches } \gamma \text{ (Match m) a } p) \wedge (\forall m \in \text{set (getNeg ms)}. \text{matches } \gamma \text{ (MatchNot (Match m)) a } p)$
 ⟨proof⟩

lemma *matches-alist-and*: $\text{matches } \gamma \text{ (alist-and l) a } p \longleftrightarrow (\forall m \in \text{set (getPos l)}. \text{matches } \gamma \text{ (Match m) a } p) \wedge (\forall m \in \text{set (getNeg l)}. \text{matches } \gamma \text{ (MatchNot (Match m)) a } p)$
 ⟨proof⟩

```

end
theory Primitive-Normalization
imports Negation-Type-Matching
begin

```

24 Primitive Normalization

24.1 Normalized Primitives

Test if a *disc* is in the match expression. For example, it call tell whether there are some matches for *Src ip*.

```
fun has-disc :: ('a ⇒ bool) ⇒ 'a match-expr ⇒ bool where
  has-disc - MatchAny = False |
  has-disc disc (Match a) = disc a |
  has-disc disc (MatchNot m) = has-disc disc m |
  has-disc disc (MatchAnd m1 m2) = (has-disc disc m1 ∨ has-disc disc m2)
```

```
fun has-disc-negated :: ('a ⇒ bool) ⇒ bool ⇒ 'a match-expr ⇒ bool where
  has-disc-negated - - MatchAny = False |
  has-disc-negated disc neg (Match a) = (if disc a then neg else False) |
  has-disc-negated disc neg (MatchNot m) = has-disc-negated disc (¬ neg) m |
  has-disc-negated disc neg (MatchAnd m1 m2) = (has-disc-negated disc neg m1 ∨
  has-disc-negated disc neg m2)
```

```
lemma ¬ has-disc-negated (λx::nat. x = 0) False (MatchAnd (Match 0) (MatchNot
(Match 1))) ⟨proof⟩
```

```
lemma has-disc-negated (λx::nat. x = 0) False (MatchAnd (Match 0) (MatchNot
(Match 0))) ⟨proof⟩
```

```
lemma has-disc-negated (λx::nat. x = 0) True (MatchAnd (Match 0) (MatchNot
(Match 1))) ⟨proof⟩
```

```
lemma ¬ has-disc-negated (λx::nat. x = 0) True (MatchAnd (Match 1) (MatchNot
(Match 0))) ⟨proof⟩
```

```
lemma has-disc-negated (λx::nat. x = 0) True (MatchAnd (Match 0) (MatchNot
(Match 0))) ⟨proof⟩
```

```
lemma has-disc-negated-MatchNot:
```

```
  has-disc-negated disc True (MatchNot m) ⟷ has-disc-negated disc False m
```

```
  has-disc-negated disc True m ⟷ has-disc-negated disc False (MatchNot m)
```

```
  ⟨proof⟩
```

```
lemma has-disc-negated-has-disc: has-disc-negated disc neg m ⟹ has-disc disc m
  ⟨proof⟩
```

```
lemma has-disc-negated-positiv-has-disc: has-disc-negated disc neg m ∨ has-disc-negated
disc (¬ neg) m ⟷ has-disc disc m
  ⟨proof⟩
```

```
lemma has-disc-negated-disj-split:
```

```
  has-disc-negated (λa. P a ∨ Q a) neg m ⟷ has-disc-negated P neg m ∨
```

```
  has-disc-negated Q neg m
```

```
  ⟨proof⟩
```

```
lemma has-disc-alist-and: has-disc disc (alist-and as) ⟷ (∃ a ∈ set as. has-disc
disc (negation-type-to-match-expr a))
```

<proof>
lemma *has-disc-negated-alist-and*: *has-disc-negated disc neg (alist-and as) \longleftrightarrow ($\exists a \in \text{set as. has-disc-negated disc neg (negation-type-to-match-expr a)$)*
 <proof>

lemma *has-disc-alist-and'*: *has-disc disc (alist-and' as) \longleftrightarrow ($\exists a \in \text{set as. has-disc disc (negation-type-to-match-expr a)$)*
 <proof>

lemma *has-disc-negated-alist-and'*: *has-disc-negated disc neg (alist-and' as) \longleftrightarrow ($\exists a \in \text{set as. has-disc-negated disc neg (negation-type-to-match-expr a)$)*
 <proof>

lemma *has-disc-alist-and'-append*:
has-disc disc' (alist-and' (ls1 @ ls2)) \longleftrightarrow
has-disc disc' (alist-and' ls1) \vee has-disc disc' (alist-and' ls2)
 <proof>

lemma *has-disc-negated-alist-and'-append*:
has-disc-negated disc' neg (alist-and' (ls1 @ ls2)) \longleftrightarrow
has-disc-negated disc' neg (alist-and' ls1) \vee has-disc-negated disc' neg (alist-and' ls2)
 <proof>

lemma *match-list-to-match-expr-not-has-disc*:
 $\forall a. \neg \text{disc } (X a) \implies \neg \text{has-disc disc (match-list-to-match-expr (map (Match \circ X) ls))}$
 <proof>

lemma *matches (($\lambda x . \text{bool-to-ternary (disc x)}$), ($\lambda - . \text{False}$)) (Match x) a p \longleftrightarrow has-disc disc (Match x)*
 <proof>

fun *normalized-n-primitive* :: (*'a \Rightarrow bool*) \times (*'a \Rightarrow 'b*) \Rightarrow (*'b \Rightarrow bool*) \Rightarrow *'a match-expr \Rightarrow bool* **where**
normalized-n-primitive - - MatchAny = True |
normalized-n-primitive (disc, sel) n (Match P) = (if disc P then n (sel P) else True) |
normalized-n-primitive (disc, sel) n (MatchNot (Match P)) = (if disc P then False else True) |
normalized-n-primitive (disc, sel) n (MatchAnd m1 m2) = (normalized-n-primitive (disc, sel) n m1 \wedge normalized-n-primitive (disc, sel) n m2) |
normalized-n-primitive - - (MatchNot (MatchAnd - -)) = False |

normalized-n-primitive - - (MatchNot (MatchNot -)) = False |
normalized-n-primitive - - (MatchNot MatchAny) = True

lemma *normalized-nnf-match-opt-MatchAny-match-expr:*
 $normalized\text{-}nnf\text{-}match\ m \implies normalized\text{-}nnf\text{-}match\ (opt\text{-}MatchAny\text{-}match\text{-}expr\ m)$
 ⟨proof⟩

lemma *normalized-n-primitive-opt-MatchAny-match-expr:*
 $normalized\text{-}n\text{-}primitive\ disc\text{-}sel\ f\ m \implies normalized\text{-}n\text{-}primitive\ disc\text{-}sel\ f\ (opt\text{-}MatchAny\text{-}match\text{-}expr\ m)$
 ⟨proof⟩

lemma *normalized-n-primitive-imp-not-disc-negated:*
 $wf\text{-}disc\text{-}sel\ (disc, sel)\ C \implies normalized\text{-}n\text{-}primitive\ (disc, sel)\ f\ m \implies \neg\ has\text{-}disc\text{-}negated\ disc\ False\ m$
 ⟨proof⟩

lemma *normalized-n-primitive-alist-and:* $normalized\text{-}n\text{-}primitive\ disc\text{-}sel\ P\ (alist\text{-}and\ as) \longleftrightarrow$
 $(\forall a \in set\ as.\ normalized\text{-}n\text{-}primitive\ disc\text{-}sel\ P\ (negation\text{-}type\text{-}to\text{-}match\text{-}expr\ a))$
 ⟨proof⟩

lemma *normalized-n-primitive-alist-and':* $normalized\text{-}n\text{-}primitive\ disc\text{-}sel\ P\ (alist\text{-}and'\ as) \longleftrightarrow$
 $(\forall a \in set\ as.\ normalized\text{-}n\text{-}primitive\ disc\text{-}sel\ P\ (negation\text{-}type\text{-}to\text{-}match\text{-}expr\ a))$
 ⟨proof⟩

lemma *not-has-disc-NegPos-map:* $\forall a.\ \neg\ disc\ (C\ a) \implies \forall a \in set\ (NegPos\text{-}map\ C\ ls).$
 $\neg\ has\text{-}disc\ disc\ (negation\text{-}type\text{-}to\text{-}match\text{-}expr\ a)$
 ⟨proof⟩

lemma *not-has-disc-negated-NegPos-map:* $\forall a.\ \neg\ disc\ (C\ a) \implies \forall a \in set\ (NegPos\text{-}map\ C\ ls).$
 $\neg\ has\text{-}disc\text{-}negated\ disc\ False\ (negation\text{-}type\text{-}to\text{-}match\text{-}expr\ a)$
 ⟨proof⟩

lemma *normalized-n-primitive-impossible-map:* $\forall a.\ \neg\ disc\ (C\ a) \implies$
 $\forall m \in set\ (map\ (Match \circ (C \circ x))\ ls).$
 $normalized\text{-}n\text{-}primitive\ (disc, sel)\ f\ m$
 ⟨proof⟩

lemma *normalized-n-primitive-alist-and'-append:*
 $normalized\text{-}n\text{-}primitive\ (disc, sel)\ f\ (alist\text{-}and'\ (ls1\ @\ ls2)) \longleftrightarrow$
 $normalized\text{-}n\text{-}primitive\ (disc, sel)\ f\ (alist\text{-}and'\ ls1) \wedge normalized\text{-}n\text{-}primitive\ (disc, sel)\ f\ (alist\text{-}and'\ ls2)$
 ⟨proof⟩

lemma *normalized-n-primitive-if-no-primitive*: $\text{normalized-nnf-match } m \implies \neg \text{has-disc } \text{disc } m \implies$
 $\text{normalized-n-primitive } (\text{disc}, \text{sel}) f m$
 ⟨proof⟩

lemma *normalized-n-primitive-false-eq-notdisc*: $\text{normalized-nnf-match } m \implies$
 $\text{normalized-n-primitive } (\text{disc}, \text{sel}) (\lambda_. \text{False}) m \longleftrightarrow \neg \text{has-disc } \text{disc } m$
 ⟨proof⟩

lemma *normalized-n-primitive-MatchAnd-combine-map*: $\text{normalized-n-primitive } \text{disc-sel } f \text{ rst} \implies$
 $\forall m' \in (\lambda \text{spt}. \text{Match } (C \text{ spt})) \text{ ' set pts. } \text{normalized-n-primitive } \text{disc-sel } f m'$
 \implies
 $m' \in (\lambda \text{spt}. \text{MatchAnd } (\text{Match } (C \text{ spt})) \text{ rst}) \text{ ' set pts} \implies \text{normalized-n-primitive } \text{disc-sel } f m'$
 ⟨proof⟩

24.2 Primitive Extractor

The following function takes a tuple of functions $((a \Rightarrow \text{bool}) \times (a \Rightarrow b))$ and a *'a match-expr*. The passed function tuple must be the discriminator and selector of the datatype package. *primitive-extractor* filters the *'a match-expr* and returns a tuple. The first element of the returned tuple is the filtered primitive matches, the second element is the remaining match expression.

It requires a *normalized-nnf-match*.

fun *primitive-extractor* :: $((a \Rightarrow \text{bool}) \times (a \Rightarrow b)) \Rightarrow 'a \text{ match-expr} \Rightarrow ('b \text{ negation-type list} \times 'a \text{ match-expr})$ **where**
primitive-extractor - *MatchAny* = $([], \text{MatchAny})$ |
primitive-extractor (*disc*,*sel*) (*Match a*) = $(\text{if } \text{disc } a \text{ then } ([\text{Pos } (\text{sel } a)], \text{MatchAny}) \text{ else } ([], \text{Match } a))$ |
primitive-extractor (*disc*,*sel*) (*MatchNot (Match a)*) = $(\text{if } \text{disc } a \text{ then } ([\text{Neg } (\text{sel } a)], \text{MatchAny}) \text{ else } ([], \text{MatchNot } (\text{Match } a)))$ |
primitive-extractor *C* (*MatchAnd ms1 ms2*) = $($
 $\text{let } (a1', ms1') = \text{primitive-extractor } C \text{ ms1};$
 $(a2', ms2') = \text{primitive-extractor } C \text{ ms2}$
 $\text{in } (a1' @ a2', \text{MatchAnd } ms1' \text{ ms2}'))$ |
primitive-extractor - - = *undefined*

The first part returned by *primitive-extractor*, here *as*: A list of primitive match expressions. For example, let $m = \text{MatchAnd } (\text{Src } ip1) (\text{Dst } ip2)$ then, using the src (*disc*, *sel*), the result is $[ip1]$. Note that *Src* is stripped from the result.

The second part, here *ms* is the match expression which was not extracted. Together, the first and second part match iff *m* matches.

lemma *primitive-extractor-fst-simp2*:

fixes $m'::'a \text{ match-expr} \Rightarrow 'a \text{ match-expr} \Rightarrow 'a \text{ match-expr}$
shows $\text{fst} (\text{case primitive-extractor } (disc, sel) m1 \text{ of } (a1', ms1') \Rightarrow \text{case primitive-extractor } (disc, sel) m2 \text{ of } (a2', ms2') \Rightarrow (a1' @ a2', m' ms1' ms2')) =$
 $\text{fst} (\text{primitive-extractor } (disc, sel) m1) @ \text{fst} (\text{primitive-extractor } (disc, sel) m2)$
 ⟨proof⟩

theorem primitive-extractor-correct: assumes
 $\text{normalized-nnf-match } m$ **and** $\text{wf-disc-sel } (disc, sel) C$ **and** $\text{primitive-extractor } (disc, sel) m = (as, ms)$
shows $\text{matches } \gamma (\text{alist-and } (NegPos-map C as)) a p \wedge \text{matches } \gamma ms a p \longleftrightarrow \text{matches } \gamma m a p$
and $\text{normalized-nnf-match } ms$
and $\neg \text{has-disc } disc ms$
and $\forall disc2. \neg \text{has-disc } disc2 m \longrightarrow \neg \text{has-disc } disc2 ms$
and $\forall disc2 sel2. \text{normalized-n-primitive } (disc2, sel2) P m \longrightarrow \text{normalized-n-primitive } (disc2, sel2) P ms$
and $\forall disc2. \neg \text{has-disc-negated } disc2 neg m \longrightarrow \neg \text{has-disc-negated } disc2 neg ms$
and $\neg \text{has-disc } disc m \longleftrightarrow as = [] \wedge ms = m$
and $\neg \text{has-disc-negated } disc False m \longleftrightarrow \text{getNeg } as = []$
and $\text{has-disc } disc m \Longrightarrow as \neq []$
 ⟨proof⟩

lemma has-disc-negated-primitive-extractor:
assumes $\text{normalized-nnf-match } m$
shows $\text{has-disc-negated } disc False m \longleftrightarrow (\exists a. Neg a \in \text{set } (\text{fst } (\text{primitive-extractor } (disc, sel) m)))$
 ⟨proof⟩

lemma primitive-extractor-reassemble-preserves:
 $\text{wf-disc-sel } (disc, sel) C \Longrightarrow$
 $\text{normalized-nnf-match } m \Longrightarrow$
 $P m \Longrightarrow$
 $P \text{ MatchAny} \Longrightarrow$
 $\text{primitive-extractor } (disc, sel) m = (as, ms) \Longrightarrow$ — turn equality around to simplify
 proof
 $(\bigwedge m1 m2. P (\text{MatchAnd } m1 m2) \longleftrightarrow P m1 \wedge P m2) \Longrightarrow$
 $(\bigwedge ls1 ls2. P (\text{alist-and}' (ls1 @ ls2)) \longleftrightarrow P (\text{alist-and}' ls1) \wedge P (\text{alist-and}' ls2))$
 \Longrightarrow
 $P (\text{alist-and}' (NegPos-map C as))$
 ⟨proof⟩

lemma primitive-extractor-reassemble-not-has-disc:
 $\text{wf-disc-sel } (disc, sel) C \Longrightarrow$
 $\text{normalized-nnf-match } m \Longrightarrow \neg \text{has-disc } disc' m \Longrightarrow$

$primitive_extractor\ (disc, sel)\ m = (as, ms) \implies$
 $\neg\ has_disc\ disc'\ (alist_and'\ (NegPos_map\ C\ as))$
 ⟨proof⟩

lemma *primitive-extractor-reassemble-not-has-disc-negated:*

$wf_disc_sel\ (disc, sel)\ C \implies$
 $normalized_nnf_match\ m \implies \neg\ has_disc_negated\ disc'\ neg\ m \implies$
 $primitive_extractor\ (disc, sel)\ m = (as, ms) \implies$
 $\neg\ has_disc_negated\ disc'\ neg\ (alist_and'\ (NegPos_map\ C\ as))$
 ⟨proof⟩

lemma *primitive-extractor-reassemble-normalized-n-primitive:*

$wf_disc_sel\ (disc, sel)\ C \implies$
 $normalized_nnf_match\ m \implies normalized_n_primitive\ (disc1, sel1)\ f\ m \implies$
 $primitive_extractor\ (disc, sel)\ m = (as, ms) \implies$
 $normalized_n_primitive\ (disc1, sel1)\ f\ (alist_and'\ (NegPos_map\ C\ as))$
 ⟨proof⟩

lemma *primitive-extractor-matchesE:* $wf_disc_sel\ (disc, sel)\ C \implies normalized_nnf_match$

$m \implies primitive_extractor\ (disc, sel)\ m = (as, ms)$
 \implies
 $(normalized_nnf_match\ ms \implies \neg\ has_disc\ disc\ ms \implies (\forall\ disc2. \neg\ has_disc\ disc2$
 $m \longrightarrow \neg\ has_disc\ disc2\ ms) \implies matches_other\ \longleftrightarrow\ matches\ \gamma\ ms\ a\ p)$
 \implies
 $matches\ \gamma\ (alist_and\ (NegPos_map\ C\ as))\ a\ p \wedge matches_other\ \longleftrightarrow\ matches\ \gamma$
 $m\ a\ p$
 ⟨proof⟩

lemma *primitive-extractor-matches-lastE:* $wf_disc_sel\ (disc, sel)\ C \implies normalized_nnf_match$

$m \implies primitive_extractor\ (disc, sel)\ m = (as, ms)$
 \implies
 $(normalized_nnf_match\ ms \implies \neg\ has_disc\ disc\ ms \implies (\forall\ disc2. \neg\ has_disc\ disc2$
 $m \longrightarrow \neg\ has_disc\ disc2\ ms) \implies matches\ \gamma\ ms\ a\ p)$
 \implies
 $matches\ \gamma\ (alist_and\ (NegPos_map\ C\ as))\ a\ p \longleftrightarrow matches\ \gamma\ m\ a\ p$
 ⟨proof⟩

The lemmas $\llbracket wf_disc_sel\ (?disc, ?sel)\ ?C; normalized_nnf_match\ ?m; primitive_extractor\ (?disc, ?sel)\ ?m = (?as, ?ms); \llbracket normalized_nnf_match\ ?ms; \neg\ has_disc\ ?disc\ ?ms; \forall\ disc2. \neg\ has_disc\ disc2\ ?m \longrightarrow \neg\ has_disc\ disc2\ ?ms \rrbracket \implies ?matches_other = matches\ ?\gamma\ ?ms\ ?a\ ?p \rrbracket \implies (matches\ ?\gamma\ (alist_and\ (NegPos_map\ ?C\ ?as))\ ?a\ ?p \wedge ?matches_other) = matches\ ?\gamma\ ?m\ ?a\ ?p$ and $\llbracket wf_disc_sel\ (?disc, ?sel)\ ?C; normalized_nnf_match\ ?m; primitive_extractor\ (?disc, ?sel)\ ?m = (?as, ?ms); \llbracket normalized_nnf_match\ ?ms; \neg\ has_disc\ ?disc\ ?ms; \forall\ disc2. \neg\ has_disc\ disc2\ ?m \longrightarrow \neg\ has_disc\ disc2\ ?ms \rrbracket \implies matches\ ?\gamma\ ?ms\ ?a\ ?p \rrbracket \implies matches\ ?\gamma\ (alist_and\ (NegPos_map\ ?C\ ?as))\ ?a\ ?p$

$= \text{matches } ?\gamma \ ?m \ ?a \ ?p$ can be used as erule to solve goals about consecutive application of *primitive-extractor*. They should be used as *primitive-extractor-matchesE[OF wf-disc-sel-for-first-extracted-thing]*.

24.3 Normalizing and Optimizing Primitives

Normalize primitives by a function f with type $'b \text{ negation-type list} \Rightarrow 'b \text{ list}$. $'b$ is a primitive type, e.g. `ipt-ipv4range`. f takes a conjunction list of negated primitives and must compress them such that:

1. no negation occurs in the output
2. the output is a disjunction of the primitives, i.e. multiple primitives in one rule are compressed to at most one primitive (leading to multiple rules)

Example with IP addresses:

```
f [10.8.0.0/16, 10.0.0.0/8] = [10.0.0.0/8]  f compresses to one range
f [10.0.0.0, 192.168.0.01] = []           range is empty, rule can be dropped
f [Neg 41] = [{0..40}, {42..ipv4max}]     one rule is translated into multiple
f [Neg 41, {20..50}, {30..50}] = [{30..40}, {42..50}]  input: conjunction list
```

definition *normalize-primitive-extract* :: $(('a \Rightarrow \text{bool}) \times ('a \Rightarrow 'b)) \Rightarrow ('b \Rightarrow 'a) \Rightarrow ('b \text{ negation-type list} \Rightarrow 'b \text{ list}) \Rightarrow 'a \text{ match-expr} \Rightarrow 'a \text{ match-expr list}$ **where**

$\text{normalize-primitive-extract } (\text{disc-sel}) \ C \ f \ m \equiv (\text{case primitive-extractor } (\text{disc-sel}) \ m \text{ of } (spts, rst) \Rightarrow \text{map } (\lambda spt. (\text{MatchAnd } (\text{Match } (C \ spt))) \ rst) \ (f \ spts))$

If f has the properties described above, then *normalize-primitive-extract* is a valid transformation of a match expression

lemma *normalize-primitive-extract*: **assumes** *normalized-nnf-match* m **and** *wf-disc-sel* disc-sel C **and**

$\forall ml. (\text{match-list } \gamma \ (\text{map } (\text{Match} \circ C) \ (f \ ml)) \ a \ p \longleftrightarrow \text{matches } \gamma \ (\text{alist-and } (\text{NegPos-map } C \ ml)) \ a \ p)$

shows *match-list* $\gamma \ (\text{normalize-primitive-extract } \text{disc-sel} \ C \ f \ m) \ a \ p \longleftrightarrow \text{matches } \gamma \ m \ a \ p$
 <proof>

thm *match-list-semantics*[$\text{of } \gamma \ (\text{map } (\text{Match} \circ C) \ (f \ ml)) \ a \ p \ [(\text{alist-and } (\text{NegPos-map } C \ ml))]]$]

corollary *normalize-primitive-extract-semantic*: **assumes** *normalized-nnf-match* *m* **and** *wf-disc-sel* *disc-sel* *C* **and**
 $\forall ml. (\text{match-list } \gamma (\text{map } (\text{Match} \circ C) (f \text{ ml})) a p \longleftrightarrow \text{matches } \gamma (\text{alist-and} (\text{NegPos-map } C \text{ ml})) a p)$
shows *approximating-bigstep-fun* $\gamma p (\text{map } (\lambda m. \text{Rule } m a) (\text{normalize-primitive-extract } \text{disc-sel } C f m)) s =$
approximating-bigstep-fun $\gamma p [\text{Rule } m a] s$
 ⟨*proof*⟩

lemma *normalize-primitive-extract-preserves-nnf-normalized*:
assumes *normalized-nnf-match* *m*
and *wf-disc-sel* (*disc*, *sel*) *C*
shows $\forall mn \in \text{set } (\text{normalize-primitive-extract } (\text{disc}, \text{sel}) C f m). \text{normalized-nnf-match } mn$
 ⟨*proof*⟩

lemma *normalize-rules-primitive-extract-preserves-nnf-normalized*:
 $\forall r \in \text{set } rs. \text{normalized-nnf-match } (\text{get-match } r) \implies \text{wf-disc-sel } \text{disc-sel } C \implies$
 $\forall r \in \text{set } (\text{normalize-rules } (\text{normalize-primitive-extract } \text{disc-sel } C f) rs). \text{normalized-nnf-match } (\text{get-match } r)$
 ⟨*proof*⟩

If something is normalized for *disc2* and *disc2* \neq *disc1* and we do something on *disc1*, then *disc2* remains normalized

lemma *normalize-primitive-extract-preserves-unrelated-normalized-n-primitive*:
assumes *normalized-nnf-match* *m*
and *normalized-n-primitive* (*disc2*, *sel2*) *P* *m*
and *wf-disc-sel* (*disc1*, *sel1*) *C*
and $\forall a. \neg \text{disc2 } (C a) \text{ — } \text{disc1 and disc2 match for different stuff. e.g. Src-Ports and Dst-Ports}$
shows $\forall mn \in \text{set } (\text{normalize-primitive-extract } (\text{disc1}, \text{sel1}) C f m). \text{normalized-n-primitive } (\text{disc2}, \text{sel2}) P mn$
 ⟨*proof*⟩

lemma *normalize-primitive-extract-normalizes-n-primitive*:
fixes *disc*::('a \Rightarrow bool) **and** *sel*::('a \Rightarrow 'b) **and** *f*::('b negation-type list \Rightarrow 'b list)
assumes *normalized-nnf-match* *m*
and *wf-disc-sel* (*disc*, *sel*) *C*
and *np*: $\forall as. (\forall a' \in \text{set } (f as). P a')$
shows $\forall m' \in \text{set } (\text{normalize-primitive-extract } (\text{disc}, \text{sel}) C f m). \text{normalized-n-primitive } (\text{disc}, \text{sel}) P m'$
 ⟨*proof*⟩

lemma *primitive-extractor-negation-type-matching1*:
assumes *wf-disc-sel* (*disc*, *sel*) *C*
and *normalized*: *normalized-nnf-match* *m*
and *a1*: *primitive-extractor* (*disc*, *sel*) *m* = (*as*, *rest*)

and *a2: matches γ m a p*
shows $(\forall m \in \text{set } (\text{map } C \text{ (getPos as)})). \text{ matches } \gamma \text{ (Match } m) \text{ } a \text{ } p) \wedge$
 $(\forall m \in \text{set } (\text{map } C \text{ (getNeg as)})). \text{ matches } \gamma \text{ (MatchNot (Match } m)) \text{ } a \text{ } p)$
<proof>

normalized-n-primitive does NOT imply *normalized-nnf-match*

lemma $\exists m. \text{ normalized-n-primitive disc-sel } f \text{ } m \longrightarrow \neg \text{ normalized-nnf-match } m$
<proof>

lemma *remove-unknowns-generic-not-has-disc: $\neg \text{ has-disc } C \text{ } m \implies \neg \text{ has-disc } C$*
(remove-unknowns-generic γ a m)
<proof>

lemma *remove-unknowns-generic-not-has-disc-negated: $\neg \text{ has-disc-negated } C \text{ } neg \text{ } m \implies \neg \text{ has-disc-negated } C \text{ } neg$*
(remove-unknowns-generic γ a m)
<proof>

lemma *remove-unknowns-generic-normalized-n-primitive: normalized-n-primitive*
disc-sel f $m \implies$
normalized-n-primitive disc-sel f (remove-unknowns-generic γ a m)
<proof>

lemma *normalize-match-preserves-disc-negated:*
shows $(\exists m\text{-DNF} \in \text{set } (\text{normalize-match } m)). \text{ has-disc-negated disc neg } m\text{-DNF}$
 $\implies \text{ has-disc-negated disc neg } m$
<proof>

has-disc-negated is a structural property and *normalize-match* is a semantical property. *normalize-match* removes subexpressions which cannot match. Thus, we cannot show (without complicated assumptions) the opposite direction of $\exists m\text{-DNF} \in \text{set } (\text{normalize-match } ?m). \text{ has-disc-negated } ?disc \text{ } ?neg \text{ } m\text{-DNF} \implies \text{ has-disc-negated } ?disc \text{ } ?neg \text{ } ?m$, because a negated primitive might occur in a subexpression which will be optimized away.

corollary *i-m-giving-this-a-funny-name-so-i-can-thank-my-future-me-when-sledgehammer-will-find-this-one-does-not*
 $\neg \text{ has-disc-negated disc neg } m \implies \forall m\text{-DNF} \in \text{set } (\text{normalize-match } m). \neg$
 $\text{ has-disc-negated disc neg } m\text{-DNF}$
<proof>

lemma *not-has-disc-opt-MatchAny-match-expr:*
 $\neg \text{ has-disc disc } m \implies \neg \text{ has-disc disc (opt-MatchAny-match-expr } m)$
<proof>

lemma *not-has-disc-negated-opt-MatchAny-match-expr:*
 $\neg \text{ has-disc-negated disc neg } m \implies \neg \text{ has-disc-negated disc neg (opt-MatchAny-match-expr } m)$

<proof>

lemma *normalize-match-preserves-nodisc:*

$\neg \text{has-disc disc } m \implies m' \in \text{set } (\text{normalize-match } m) \implies \neg \text{has-disc disc } m'$
<proof>

lemma *not-has-disc-normalize-match:*

$\neg \text{has-disc-negated disc neg } m \implies m' \in \text{set } (\text{normalize-match } m) \implies \neg \text{has-disc-negated disc neg } m'$
<proof>

lemma *normalize-match-preserves-normalized-n-primitive:*

normalized-n-primitive disc-sel f rst \implies
 $\forall m \in \text{set } (\text{normalize-match } rst). \text{normalized-n-primitive disc-sel f } m$
<proof>

24.4 Optimizing a match expression

Optimizes a match expression with a function that takes *'b negation-type list* and returns *('b list × 'b list) option*. The function should return *None* if the match expression cannot match. It returns *Some (as-pos, as-neg)* where *as-pos* and *as-neg* are lists of primitives. Positive and Negated. The result is one match expression.

In contrast *normalize-primitive-extract* returns a list of match expression, to be read as their disjunction.

definition *compress-normalize-primitive* :: $((\text{'a} \Rightarrow \text{bool}) \times (\text{'a} \Rightarrow \text{'b})) \Rightarrow (\text{'b} \Rightarrow \text{'a}) \Rightarrow$
 $(\text{'b negation-type list} \Rightarrow (\text{'b list} \times \text{'b list})$
option) ⇒
 $\text{'a match-expr} \Rightarrow \text{'a match-expr option}$ **where**
compress-normalize-primitive disc-sel C f m \equiv *(case primitive-extractor disc-sel*
m of (as, rst) ⇒
 $(\text{map-option } (\lambda(\text{as-pos}, \text{as-neg}). \text{MatchAnd}$
 $(\text{alist-and}' (\text{NegPos-map } C ((\text{map Pos as-pos})@(\text{map}$
Neg as-neg))))
 rst
 $) (f \text{ as}))$

lemma *compress-normalize-primitive-nnf: wf-disc-sel disc-sel C* \implies
normalized-nnf-match m \implies *compress-normalize-primitive disc-sel C f m =*
Some m' ⇒
normalized-nnf-match m'
<proof>

lemma *compress-normalize-primitive-not-introduces-C:*

assumes *notdisc*: \neg *has-disc* *disc* *m*
and *wf*: *wf-disc-sel* (*disc*,*sel*) *C'*
and *nm*: *normalized-nnf-match* *m*
and *some*: *compress-normalize-primitive* (*disc*,*sel*) *C* *f* *m* = *Some* *m'*
and *f-preserves*: \bigwedge *as-pos* *as-neg*. *f* [] = *Some* (*as-pos*, *as-neg*) \implies *as-pos* =
[] \wedge *as-neg* = []
shows \neg *has-disc* *disc* *m'*
<*proof*>

lemma *compress-normalize-primitive-not-introduces-C-negated*:
assumes *notdisc*: \neg *has-disc-negated* *disc* *False* *m*
and *wf*: *wf-disc-sel* (*disc*,*sel*) *C*
and *nm*: *normalized-nnf-match* *m*
and *some*: *compress-normalize-primitive* (*disc*,*sel*) *C* *f* *m* = *Some* *m'*
and *f-preserves*: \bigwedge *as* *as-pos* *as-neg*. *f* *as* = *Some* (*as-pos*, *as-neg*) \implies *getNeg*
as = [] \implies *as-neg* = []
shows \neg *has-disc-negated* *disc* *False* *m'*
<*proof*>

lemma *compress-normalize-primitive-Some*:
assumes *normalized*: *normalized-nnf-match* *m*
and *wf*: *wf-disc-sel* (*disc*,*sel*) *C*
and *some*: *compress-normalize-primitive* (*disc*,*sel*) *C* *f* *m* = *Some* *m'*
and *f-correct*: \bigwedge *as* *as-pos* *as-neg*. *f* *as* = *Some* (*as-pos*, *as-neg*) \implies
matches γ (*alist-and* (*NegPos-map* *C* ((*map* *Pos* *as-pos*)@(*map* *Neg*
as-neg)))) *a* *p* \longleftrightarrow
matches γ (*alist-and* (*NegPos-map* *C* *as*)) *a* *p*
shows *matches* γ *m'* *a* *p* \longleftrightarrow *matches* γ *m* *a* *p*
<*proof*>

lemma *compress-normalize-primitive-None*:
assumes *normalized*: *normalized-nnf-match* *m*
and *wf*: *wf-disc-sel* (*disc*,*sel*) *C*
and *none*: *compress-normalize-primitive* (*disc*,*sel*) *C* *f* *m* = *None*
and *f-correct*: \bigwedge *as*. *f* *as* = *None* \implies \neg *matches* γ (*alist-and* (*NegPos-map* *C*
as)) *a* *p*
shows \neg *matches* γ *m* *a* *p*
<*proof*>

lemma *compress-normalize-primitive-hasdisc*:
assumes *am*: \neg *has-disc* *disc2* *m*
and *wf*: *wf-disc-sel* (*disc*,*sel*) *C*

and *disc*: $(\forall a. \neg \text{disc2 } (C a))$
and *nm*: *normalized-nnf-match* *m*
and *some*: *compress-normalize-primitive* (*disc*,*sel*) *C f m* = *Some m'*
shows *normalized-nnf-match* *m' ∧ ¬ has-disc disc2 m'*
 ⟨*proof*⟩

lemma *compress-normalize-primitive-hasdisc-negated*:

assumes *am*: $\neg \text{has-disc-negated } \text{disc2 } \text{neg } m$
and *wf*: *wf-disc-sel* (*disc*,*sel*) *C*
and *disc*: $(\forall a. \neg \text{disc2 } (C a))$
and *nm*: *normalized-nnf-match* *m*
and *some*: *compress-normalize-primitive* (*disc*,*sel*) *C f m* = *Some m'*
shows *normalized-nnf-match* *m' ∧ ¬ has-disc-negated disc2 neg m'*
 ⟨*proof*⟩

thm *normalize-primitive-extract-preserves-unrelated-normalized-n-primitive*

lemma *compress-normalize-primitive-preserves-normalized-n-primitive*:

assumes *am*: *normalized-n-primitive* (*disc2*, *sel2*) *P m*
and *wf*: *wf-disc-sel* (*disc*,*sel*) *C*
and *disc*: $(\forall a. \neg \text{disc2 } (C a))$
and *nm*: *normalized-nnf-match* *m*
and *some*: *compress-normalize-primitive* (*disc*,*sel*) *C f m* = *Some m'*
shows *normalized-nnf-match* *m' ∧ normalized-n-primitive* (*disc2*, *sel2*) *P m'*
 ⟨*proof*⟩

24.5 Processing a list of normalization functions

fun *compress-normalize-primitive-monad* :: (*'a match-expr* ⇒ *'a match-expr option*) *list* ⇒ *'a match-expr* ⇒ *'a match-expr option* **where**
compress-normalize-primitive-monad [] *m* = *Some m* |
compress-normalize-primitive-monad (*f#fs*) *m* = (case *f m* of *None* ⇒ *None*
 | *Some m'* ⇒
compress-normalize-primitive-monad fs m')

lemma *compress-normalize-primitive-monad*:

assumes $\bigwedge m m' f. f \in \text{set } fs \implies \text{normalized-nnf-match } m \implies f m = \text{Some } m'$
 $m' \implies \text{matches } \gamma m' a p \iff \text{matches } \gamma m a p$
and $\bigwedge m m' f. f \in \text{set } fs \implies \text{normalized-nnf-match } m \implies f m = \text{Some } m'$
 $m' \implies \text{normalized-nnf-match } m'$
and *normalized-nnf-match* *m*
and (*compress-normalize-primitive-monad* *fs m*) = *Some m'*
shows *matches* $\gamma m' a p \iff \text{matches } \gamma m a p$ (**is** *?goal1*)
and *normalized-nnf-match* *m'* (**is** *?goal2*)
 ⟨*proof*⟩

lemma *compress-normalize-primitive-monad-None*:

assumes $\bigwedge m m' f. f \in \text{set } fs \implies \text{normalized-nnf-match } m \implies f m = \text{Some } m'$
 $m' \implies \text{matches } \gamma m' a p \iff \text{matches } \gamma m a p$

```

    and  $\bigwedge m f. f \in \text{set } fs \implies \text{normalized-nnf-match } m \implies f m = \text{None} \implies$ 
 $\neg \text{matches } \gamma m a p$ 
    and  $\bigwedge m m' f. f \in \text{set } fs \implies \text{normalized-nnf-match } m \implies f m = \text{Some}$ 
 $m' \implies \text{normalized-nnf-match } m'$ 
    and  $\text{normalized-nnf-match } m$ 
    and  $(\text{compress-normalize-primitive-monad } fs m) = \text{None}$ 
    shows  $\neg \text{matches } \gamma m a p$ 
    <proof>

```

lemma *compress-normalize-primitive-monad-preserves:*

```

    assumes  $\bigwedge m m' f. f \in \text{set } fs \implies \text{normalized-nnf-match } m \implies f m = \text{Some}$ 
 $m' \implies \text{normalized-nnf-match } m'$ 
    and  $\bigwedge m m' f. f \in \text{set } fs \implies \text{normalized-nnf-match } m \implies P m \implies f m$ 
 $= \text{Some } m' \implies P m'$ 
    and  $\text{normalized-nnf-match } m$ 
    and  $P m$ 
    and  $(\text{compress-normalize-primitive-monad } fs m) = \text{Some } m'$ 
    shows  $\text{normalized-nnf-match } m' \wedge P m'$ 
    <proof>

```

```

datatype 'a match-compress = CannotMatch | MatchesAll | MatchExpr 'a

```

```

end

```

25 Combine Match Expressions

```

theory MatchExpr-Fold

```

```

imports Primitive-Normalization

```

```

begin

```

```

fun andfold-MatchExp :: 'a match-expr list  $\Rightarrow$  'a match-expr where

```

```

    andfold-MatchExp [] = MatchAny |

```

```

    andfold-MatchExp [e] = e |

```

```

    andfold-MatchExp (e#es) = MatchAnd e (andfold-MatchExp es)

```

```

lemma andfold-MatchExp-alist-and: alist-and' (map Pos ls) = andfold-MatchExp
(map Match ls)

```

```

    <proof>

```

```

lemma andfold-MatchExp-matches:

```

```

    matches  $\gamma$  (andfold-MatchExp ms) a p  $\iff$   $(\forall m \in \text{set } ms. \text{matches } \gamma m a p)$ 

```

```

    <proof>

```

lemma *andfold-MatchExp-not-discI*:
 $\forall m \in \text{set } ms. \neg \text{has-disc disc } m \implies \neg \text{has-disc disc } (\text{andfold-MatchExp } ms)$
 $\langle \text{proof} \rangle$

lemma *andfold-MatchExp-not-disc-negatedI*:
 $\forall m \in \text{set } ms. \neg \text{has-disc-negated disc neg } m \implies \neg \text{has-disc-negated disc neg } (\text{andfold-MatchExp } ms)$
 $\langle \text{proof} \rangle$

lemma *andfold-MatchExp-not-disc-negated-mapMatch*:
 $\neg \text{has-disc-negated disc False } (\text{andfold-MatchExp } (\text{map } (\text{Match } \circ C) \text{ ls}))$
 $\langle \text{proof} \rangle$

lemma *andfold-MatchExp-not-disc-mapMatch*:
 $\forall a. \neg \text{disc } (C a) \implies \neg \text{has-disc disc } (\text{andfold-MatchExp } (\text{map } (\text{Match } \circ C) \text{ ls}))$
 $\langle \text{proof} \rangle$

lemma *andfold-MatchExp-normalized-nnf*: $\forall m \in \text{set } ms. \text{normalized-nnf-match } m \implies \text{normalized-nnf-match } (\text{andfold-MatchExp } ms)$
 $\langle \text{proof} \rangle$

lemma *andfold-MatchExp-normalized-n-primitive*: $\forall m \in \text{set } ms. \text{normalized-n-primitive } (\text{disc, sel}) f m \implies \text{normalized-n-primitive } (\text{disc, sel}) f (\text{andfold-MatchExp } ms)$
 $\langle \text{proof} \rangle$

lemma *andfold-MatchExp-normalized-normalized-n-primitive-single*:
 $\forall a. \neg \text{disc } (C a) \implies s \in \text{set } (\text{normalize-match } (\text{andfold-MatchExp } (\text{map } (\text{Match } \circ C) \text{ xs}))) \implies \text{normalized-n-primitive } (\text{disc, sel}) f s$
 $\langle \text{proof} \rangle$

lemma *normalize-andfold-MatchExp-normalized-n-primitive*:
 $\forall m \in \text{set } ms. \forall s' \in \text{set } (\text{normalize-match } m). \text{normalized-n-primitive } (\text{disc, sel}) f s' \implies s \in \text{set } (\text{normalize-match } (\text{andfold-MatchExp } ms)) \implies \text{normalized-n-primitive } (\text{disc, sel}) f s$
 $\langle \text{proof} \rangle$

end

theory *Common-Primitive-Lemmas*

imports *Common-Primitive-Matcher*
../Semantics-Ternary/Primitive-Normalization
../Semantics-Ternary/MatchExpr-Fold

begin

26 Further Lemmas about the Common Matcher

lemma *has-unknowns-common-matcher*: **fixes** $m::'i::\text{len common-primitive match-expr}$

shows *has-unknowns common-matcher m* \longleftrightarrow *has-disc is-Extra m*
 ⟨*proof*⟩

end
theory *Ports-Normalize*
imports *Common-Primitive-Lemmas*
begin

27 Normalizing L4 Ports

27.1 Defining Normalized Ports

fun *normalized-src-ports* :: '*i*::*len* common-primitive match-expr \Rightarrow bool **where**
normalized-src-ports *MatchAny* = *True* |
normalized-src-ports (*Match* (*Src-Ports* (*L4Ports* - []))) = *True* |
normalized-src-ports (*Match* (*Src-Ports* (*L4Ports* - [-]))) = *True* |
normalized-src-ports (*Match* (*Src-Ports* -)) = *False* |
normalized-src-ports (*Match* -) = *True* |
normalized-src-ports (*MatchNot* (*Match* (*Src-Ports* -))) = *False* |
normalized-src-ports (*MatchNot* (*Match* -)) = *True* |
normalized-src-ports (*MatchAnd* *m1* *m2*) = (*normalized-src-ports* *m1* \wedge *normalized-src-ports* *m2*) |
normalized-src-ports (*MatchNot* (*MatchAnd* - -)) = *False* |
normalized-src-ports (*MatchNot* (*MatchNot* -)) = *False* |
normalized-src-ports (*MatchNot* *MatchAny*) = *True*

fun *normalized-dst-ports* :: '*i*::*len* common-primitive match-expr \Rightarrow bool **where**
normalized-dst-ports *MatchAny* = *True* |
normalized-dst-ports (*Match* (*Dst-Ports* (*L4Ports* - []))) = *True* |
normalized-dst-ports (*Match* (*Dst-Ports* (*L4Ports* - [-]))) = *True* |
normalized-dst-ports (*Match* (*Dst-Ports* -)) = *False* |
normalized-dst-ports (*Match* -) = *True* |
normalized-dst-ports (*MatchNot* (*Match* (*Dst-Ports* -))) = *False* |
normalized-dst-ports (*MatchNot* (*Match* -)) = *True* |
normalized-dst-ports (*MatchAnd* *m1* *m2*) = (*normalized-dst-ports* *m1* \wedge *normalized-dst-ports* *m2*) |
normalized-dst-ports (*MatchNot* (*MatchAnd* - -)) = *False* |
normalized-dst-ports (*MatchNot* (*MatchNot* -)) = *False* |
normalized-dst-ports (*MatchNot* *MatchAny*) = *True*

lemma *normalized-src-ports-def2*: *normalized-src-ports* *ms* = *normalized-n-primitive* (*is-Src-Ports*, *src-ports-sel*) (λ *ps*. *case* *ps* of *L4Ports* - *pts* \Rightarrow *length* *pts* \leq 1) *ms*
 ⟨*proof*⟩

lemma *normalized-dst-ports-def2*: *normalized-dst-ports* *ms* = *normalized-n-primitive* (*is-Dst-Ports*, *dst-ports-sel*) (λ *ps*. *case* *ps* of *L4Ports* - *pts* \Rightarrow *length* *pts* \leq 1) *ms*
 ⟨*proof*⟩

Idea: first, remove all negated matches, then *normalize-match*, then only work with *primitive-extractor* on *Pos* ones. They only need an intersect and split later on.

This is not very efficient because normalizing nnf will blow up a lot. but we can tune performance later on go for correctness first! Anything with *MatchOr* and *normalize-match* later is a bit inefficient.

27.2 Compressing Positive Matches on Ports into a Single Match

```

fun l4-ports-compress :: ipt-l4-ports list  $\Rightarrow$  ipt-l4-ports match-compress where
  l4-ports-compress [] = MatchesAll |
  l4-ports-compress [L4Ports proto ps] = MatchExpr (L4Ports proto (wi2l (wordinterval-compress
(l2wi ps)))) |
  l4-ports-compress (L4Ports proto1 ps1 # L4Ports proto2 ps2 # pss) =
    (if
      proto1  $\neq$  proto2
    then
      CannotMatch
    else
      l4-ports-compress (L4Ports proto1 (wi2l (wordinterval-intersection (l2wi
ps1) (l2wi ps2)))) # pss
    )

value[code] l4-ports-compress [L4Ports TCP [(22,22), (23,23)]]

```

```

lemma raw-ports-compress-src-CannotMatch:
fixes p :: ('i::len, 'a) tagged-packet-scheme
assumes generic: primitive-matcher-generic  $\beta$ 
and c: l4-ports-compress pss = CannotMatch
shows  $\neg$  matches ( $\beta$ ,  $\alpha$ ) (alist-and (map (Pos  $\circ$  Src-Ports) pss)) a p
<proof>

```

```

lemma raw-ports-compress-dst-CannotMatch:
fixes p :: ('i::len, 'a) tagged-packet-scheme
assumes generic: primitive-matcher-generic  $\beta$ 
and c: l4-ports-compress pss = CannotMatch
shows  $\neg$  matches ( $\beta$ ,  $\alpha$ ) (alist-and (map (Pos  $\circ$  Dst-Ports) pss)) a p
<proof>

```

```

lemma l4-ports-compress-length-Matchall: length pss > 0  $\implies$  l4-ports-compress
pss  $\neq$  MatchesAll
<proof>

```

```

lemma raw-ports-compress-MatchesAll:
fixes p :: ('i::len, 'a) tagged-packet-scheme
assumes generic: primitive-matcher-generic  $\beta$ 

```

and $c: l_4\text{-ports-compress } pss = \text{MatchesAll}$
shows $\text{matches } (\beta, \alpha) (\text{alist-and } (\text{map } (\text{Pos} \circ \text{Src-Ports}) pss)) a p$
and $\text{matches } (\beta, \alpha) (\text{alist-and } (\text{map } (\text{Pos} \circ \text{Dst-Ports}) pss)) a p$
 $\langle \text{proof} \rangle$

lemma $\text{raw-ports-compress-src-MatchExpr}$:
fixes $p :: ('i::\text{len}, 'a) \text{tagged-packet-scheme}$
assumes $\text{generic: primitive-matcher-generic } \beta$
and $c: l_4\text{-ports-compress } pss = \text{MatchExpr } m$
shows $\text{matches } (\beta, \alpha) (\text{Match } (\text{Src-Ports } m)) a p \longleftrightarrow \text{matches } (\beta, \alpha) (\text{alist-and } (\text{map } (\text{Pos} \circ \text{Src-Ports}) pss)) a p$
 $\langle \text{proof} \rangle$

lemma $\text{raw-ports-compress-dst-MatchExpr}$:
fixes $p :: ('i::\text{len}, 'a) \text{tagged-packet-scheme}$
assumes $\text{generic: primitive-matcher-generic } \beta$
and $c: l_4\text{-ports-compress } pss = \text{MatchExpr } m$
shows $\text{matches } (\beta, \alpha) (\text{Match } (\text{Dst-Ports } m)) a p \longleftrightarrow \text{matches } (\beta, \alpha) (\text{alist-and } (\text{map } (\text{Pos} \circ \text{Dst-Ports}) pss)) a p$
 $\langle \text{proof} \rangle$

27.3 Rewriting Negated Matches on Ports

fun $l_4\text{-ports-negate-one}$
 $:: (\text{ipt-}l_4\text{-ports} \Rightarrow 'i \text{common-primitive}) \Rightarrow \text{ipt-}l_4\text{-ports} \Rightarrow ('i::\text{len} \text{common-primitive})$
 match-expr
where
 $l_4\text{-ports-negate-one } C (\text{L4Ports } \text{proto } \text{pts}) = \text{MatchOr}$
 $(\text{MatchNot } (\text{Match } (\text{Prot } (\text{Proto } \text{proto}))))$
 $(\text{Match } (C (\text{L4Ports } \text{proto } (\text{raw-ports-invert } \text{pts}))))$

lemma $l_4\text{-ports-negate-one}$:
fixes $p :: ('i::\text{len}, 'a) \text{tagged-packet-scheme}$
assumes $\text{generic: primitive-matcher-generic } \beta$
shows $\text{matches } (\beta, \alpha) (l_4\text{-ports-negate-one } \text{Src-Ports } \text{ports}) a p \longleftrightarrow$
 $\text{matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{Src-Ports } \text{ports}))) a p$
and $\text{matches } (\beta, \alpha) (l_4\text{-ports-negate-one } \text{Dst-Ports } \text{ports}) a p \longleftrightarrow$
 $\text{matches } (\beta, \alpha) (\text{MatchNot } (\text{Match } (\text{Dst-Ports } \text{ports}))) a p$
 $\langle \text{proof} \rangle$

lemma $l_4\text{-ports-negate-one-nodisc}$:
 $\forall a. \neg \text{disc } (C a) \implies \forall a. \neg \text{disc } (\text{Prot } a) \implies \neg \text{has-disc disc } (l_4\text{-ports-negate-one } C \text{ pt})$
 $\langle \text{proof} \rangle$

lemma $l_4\text{-ports-negate-one-not-has-disc-negated-generic}$:
assumes $\text{noProt: } \forall a. \neg \text{disc } (\text{Prot } a)$
shows $\neg \text{has-disc-negated disc False } (l_4\text{-ports-negate-one } C \text{ ports})$
 $\langle \text{proof} \rangle$

lemma *l4-ports-negate-one-not-has-disc-negated*:
 $\neg \text{has-disc-negated is-Src-Ports False (l4-ports-negate-one Src-Ports ports)}$
 $\neg \text{has-disc-negated is-Dst-Ports False (l4-ports-negate-one Dst-Ports ports)}$
 $\langle \text{proof} \rangle$

lemma *negated-normalized-folded-ports-nodisc*:
 $\forall a. \neg \text{disc (C a)} \implies (\forall a. \neg \text{disc (Prot a)}) \vee \text{pts} = [] \implies$
 $m \in \text{set (normalize-match (andfold-MatchExp (map (l4-ports-negate-one C)$
 $\text{pts})) \implies$
 $\neg \text{has-disc disc m}$
 $\langle \text{proof} \rangle$

lemma *negated-normalized-folded-ports-normalized-n-primitive*:
 $\forall a. \neg \text{disc (C a)} \implies (\forall a. \neg \text{disc (Prot a)}) \vee \text{pts} = [] \implies$
 $x \in \text{set (normalize-match (andfold-MatchExp (map (l4-ports-negate-one C)$
 $\text{pts})) \implies$
 $\text{normalized-n-primitive (disc, sel) f x}$
 $\langle \text{proof} \rangle$

beware, the result is not nnf normalized!

lemma $\neg \text{normalized-nnf-match (l4-ports-negate-one C ports)}$
 $\langle \text{proof} \rangle$

Warning: does not preserve negated primitive property in general. Might be violated for *Prot*. We will nnf normalize after applying the function.

lemma $\forall a. \neg \text{disc (C a)} \implies \neg \text{normalized-n-primitive (disc, sel) f (l4-ports-negate-one C a)}$
 $\langle \text{proof} \rangle$

declare *l4-ports-negate-one.simps[simp del]*

lemma $((\text{normalize-match (l4-ports-negate-one Src-Ports (L4Ports TCP [(22,22),(80,90)]))})::$
 $32 \text{ common-primitive match-expr list})$
 $=$
 $[\text{MatchNot (Match (Prot (Proto TCP)))}$
 $, \text{Match (Src-Ports (L4Ports 6 [(0, 21), (23, 79), (91, 0xFFFF)]))}] \langle \text{proof} \rangle$

definition *rewrite-negated-primitives*
 $:: ((\text{'a} \Rightarrow \text{bool}) \times (\text{'a} \Rightarrow \text{'b})) \Rightarrow (\text{'b} \Rightarrow \text{'a}) \Rightarrow \text{— disc-sel C}$
 $((\text{'b} \Rightarrow \text{'a}) \Rightarrow \text{'b} \Rightarrow \text{'a match-expr}) \Rightarrow \text{— negate-one function}$
 $\text{'a match-expr} \Rightarrow \text{'a match-expr}$ **where**
 $\text{rewrite-negated-primitives disc-sel C negate m} \equiv$
 $\text{let (spts, rst) = primitive-extractor disc-sel m}$
 $\text{in if getNeg spts} = [] \text{ then m else}$
 MatchAnd
 $(\text{andfold-MatchExp (map (negate C) (getNeg spts))})$

$(MatchAnd$
 $(andfold-MatchExp (map (Match \circ C) (getPos\ pts))))$ — TODO:
 compress all the positive ports into one?
 $rst)$

It does nothing if there is not even a negated primitive in it

lemma *rewrite-negated-primitives-unchanged-if-not-has-disc-negated:*
assumes n : *normalized-nnf-match* m
and $wf-disc-sel$: *wf-disc-sel* $(disc, sel)$ C
and $noDisc$: \neg *has-disc-negated* $disc$ *False* m
shows *rewrite-negated-primitives* $(disc, sel)$ C *negate-f* $m = m$
 $\langle proof \rangle$

lemma *rewrite-negated-primitives-normalized-no-modification:*
assumes $wf-disc-sel$: *wf-disc-sel* $(disc, sel)$ C
and $disc-p$: \neg *has-disc-negated* $disc$ *False* m
and n : *normalized-nnf-match* m
and a : $a \in set$ $(normalize-match (rewrite-negated-primitives (disc, sel) C$
 $l4$ -ports-negate-one $m))$
shows $a = m$
 $\langle proof \rangle$

lemma *rewrite-negated-primitives-preserves-not-has-disc:*
assumes n : *normalized-nnf-match* m
and $wf-disc-sel$: *wf-disc-sel* $(disc, sel)$ C
and $nodisc$: \neg *has-disc* $disc2$ m
and $noNeg$: \neg *has-disc-negated* $disc$ *False* m
and $disc2-noC$: $\forall a. \neg disc2 (C a)$
shows \neg *has-disc* $disc2$ $(rewrite-negated-primitives (disc, sel) C$ $l4$ -ports-negate-one
 $m)$
 $\langle proof \rangle$

lemma *rewrite-negated-primitives:*
assumes n : *normalized-nnf-match* m **and** $wf-disc-sel$: *wf-disc-sel* $disc-sel$ C
and $negate-f$: $\forall pts. matches \gamma (negate-f C pts) a p \iff matches \gamma (MatchNot$
 $(Match (C pts))) a p$
shows $matches \gamma (rewrite-negated-primitives disc-sel C negate-f m) a p \iff$
 $matches \gamma m a p$
 $\langle proof \rangle$

lemma *rewrite-negated-primitives-not-has-disc:*
assumes n : *normalized-nnf-match* m **and** $wf-disc-sel$: *wf-disc-sel* $(disc, sel)$ C
and $nodisc$: \neg *has-disc* $disc2$ m
and $negate-f$: *has-disc-negated* $disc$ *False* $m \implies \forall pts. \neg$ *has-disc* $disc2$ $(negate-f$
 $C pts)$
and $no-disc$: $\forall a. \neg disc2 (C a)$
shows \neg *has-disc* $disc2$ $(rewrite-negated-primitives (disc, sel) C negate-f m)$

<proof>

lemma *rewrite-negated-primitives-not-has-disc-negated:*

assumes *n: normalized-nnf-match m and wf-disc-sel: wf-disc-sel (disc,sel) C*

and *negate-f: has-disc-negated disc False m $\implies \forall pts. \neg has-disc-negated disc False (negate-f C pts)$*

shows *$\neg has-disc-negated disc False (rewrite-negated-primitives (disc,sel) C negate-f m)$*

<proof>

lemma *rewrite-negated-primitives-preserves-not-has-disc-negated:*

assumes *n: normalized-nnf-match m and wf-disc-sel: wf-disc-sel (disc,sel) C*

and *negate-f: has-disc-negated disc False m $\implies \forall pts. \neg has-disc-negated disc2 False (negate-f C pts)$*

and *no-disc: $\neg has-disc-negated disc2 False m$*

shows *$\neg has-disc-negated disc2 False (rewrite-negated-primitives (disc,sel) C negate-f m)$*

<proof>

lemma *rewrite-negated-primitives-normalized-preserves-unrelated-helper:*

assumes *wf-disc-sel: wf-disc-sel (disc, sel) C*

and *disc: $\forall a. \neg disc2 (C a)$*

and *disc-p: $(\forall a. \neg disc2 (Prot a)) \vee \neg has-disc-negated disc False m$*

shows *normalized-nnf-match m \implies*

normalized-n-primitive (disc2, sel2) f m \implies

$a \in set (normalize-match (rewrite-negated-primitives (disc, sel) C l4-ports-negate-one m)) \implies$

normalized-n-primitive (disc2, sel2) f a

<proof>

definition *rewrite-negated-src-ports*

:: 'i::len common-primitive match-expr \Rightarrow 'i common-primitive match-expr

where

rewrite-negated-src-ports m \equiv

rewrite-negated-primitives (is-Src-Ports, src-ports-sel) Src-Ports l4-ports-negate-one m

definition *rewrite-negated-dst-ports*

:: 'i::len common-primitive match-expr \Rightarrow 'i common-primitive match-expr

where

rewrite-negated-dst-ports m \equiv

rewrite-negated-primitives (is-Dst-Ports, dst-ports-sel) Dst-Ports l4-ports-negate-one m

value *rewrite-negated-src-ports (MatchAnd (Match (Dst (IpAddrNetmask (ipv4addr-of-dotdecimal (127, 0, 0, 0)) 8)))*

(MatchAnd (Match (Prot (Proto TCP))))

```

    (MatchNot (Match (Src-Ports (L4Ports UDP [(80,80)]))))
  ))
value rewrite-negated-src-ports (MatchAnd (Match (Dst (IpAddrNetmask (ipv4addr-of-dotdecimal
(127, 0, 0, 0)) 8)))
  (MatchAnd (Match (Prot (Proto TCP)))
    (MatchNot (Match (Extra "foobar"))))
  ))

```

lemma *rewrite-negated-src-ports:*

assumes *generic: primitive-matcher-generic* β **and** *n: normalized-nnf-match* m

shows *matches* (β, α) (*rewrite-negated-src-ports* m) a $p \longleftrightarrow$ *matches* (β, α) m
 a p
 \langle *proof* \rangle

lemma *rewrite-negated-dst-ports:*

assumes *generic: primitive-matcher-generic* β **and** *n: normalized-nnf-match* m

shows *matches* (β, α) (*rewrite-negated-dst-ports* m) a $p \longleftrightarrow$ *matches* (β, α) m
 a p
 \langle *proof* \rangle

lemma *rewrite-negated-src-ports-not-has-disc-negated:*

assumes *n: normalized-nnf-match* m

shows \neg *has-disc-negated is-Src-Ports False* (*rewrite-negated-src-ports* m)
 \langle *proof* \rangle

lemma *rewrite-negated-dst-ports-not-has-disc-negated:*

assumes *n: normalized-nnf-match* m

shows \neg *has-disc-negated is-Dst-Ports False* (*rewrite-negated-dst-ports* m)
 \langle *proof* \rangle

lemma \neg *has-disc-negated disc t m* $\implies \forall m' \in$ *set (normalize-match m)*. \neg
has-disc-negated disc t m'
 \langle *proof* \rangle

corollary *normalize-rewrite-negated-src-ports-not-has-disc-negated:*

assumes *n: normalized-nnf-match* m

shows $\forall m' \in$ *set (normalize-match (rewrite-negated-src-ports m))*. \neg *has-disc-negated is-Src-Ports False* m'
 \langle *proof* \rangle

27.4 Normalizing Positive Matches on Ports

fun *singletonize-L4Ports* :: *ipt-l4-ports* \Rightarrow *ipt-l4-ports list* **where**

singletonize-L4Ports (*L4Ports proto pts*) = *map* ($\lambda p.$ *L4Ports proto* [p]) *pts*

lemma *singletonize-L4Ports-src:* **assumes** *generic: primitive-matcher-generic* β

shows *match-list* (β, α) (*map* (*Match* \circ *Src-Ports*) (*singletonize-L4Ports pts*))

$a\ p \longleftrightarrow$
matches (β, α) (*Match* (*Src-Ports pts*)) $a\ p$
 ⟨*proof*⟩

lemma *singletonize-L4Ports-dst: assumes generic: primitive-matcher-generic* β
shows *match-list* (β, α) (*map* (*Match* \circ *Dst-Ports*) (*singletonize-L4Ports pts*))

$a\ p \longleftrightarrow$
matches (β, α) (*Match* (*Dst-Ports pts*)) $a\ p$
 ⟨*proof*⟩

lemma *singletonize-L4Ports-normalized-generic:*

assumes *wf-disc-sel: wf-disc-sel* (*disc, sel*) C

and $m' \in (\lambda spt. \text{Match } (C\ spt))$ ‘*set* (*singletonize-L4Ports pt*)

shows *normalized-n-primitive* (*disc, sel*) (*case-ipt-l4-ports* ($\lambda x\ pts. \text{length } pts \leq 1$)) m'
 ⟨*proof*⟩

lemma *singletonize-L4Ports-normalized-src-ports:*

$m' \in (\lambda spt. \text{Match } (Src-Ports\ spt))$ ‘*set* (*singletonize-L4Ports pt*) \implies *normalized-src-ports* m'
 ⟨*proof*⟩

lemma *singletonize-L4Ports-normalized-dst-ports:*

$m' \in (\lambda spt. \text{Match } (Dst-Ports\ spt))$ ‘*set* (*singletonize-L4Ports pt*) \implies *normalized-dst-ports* m'
 ⟨*proof*⟩

declare *singletonize-L4Ports.simps*[*simp del*]

lemma *normalized-ports-singletonize-combine-rst:*

assumes *wf-disc-sel: wf-disc-sel* (*disc, sel*) C

shows *normalized-n-primitive* (*disc, sel*) (*case-ipt-l4-ports* ($\lambda x\ pts. \text{length } pts \leq 1$)) \implies
 $m' \in (\lambda spt. \text{MatchAnd } (\text{Match } (C\ spt))\ rst)$ ‘*set* (*singletonize-L4Ports pt*) \implies
normalized-n-primitive (*disc, sel*) (*case-ipt-l4-ports* ($\lambda x\ pts. \text{length } pts \leq 1$)) m'
 ⟨*proof*⟩

Normalizing match expressions such that at most one port will exist in it.
 Returns a list of match expressions (splits one firewall rule into several rules).

definition *normalize-positive-ports-step*

$:: ((i::\text{len } \text{common-primitive} \implies \text{bool}) \times (i\ \text{common-primitive} \implies \text{ipt-l4-ports}))$
 \implies
 $(\text{ipt-l4-ports} \implies i\ \text{common-primitive}) \implies$
 $i\ \text{common-primitive } \text{match-expr} \implies i\ \text{common-primitive } \text{match-expr list}$

where

normalize-positive-ports-step disc-sel C m \equiv

let (*spts, rst*) = *primitive-extractor disc-sel m in*
case (*getPos spts, getNeg spts*)

of (pspts, []) \Rightarrow (case l4-ports-compress pspts of CannotMatch \Rightarrow []
| MatchesAll \Rightarrow [rst]
| MatchExpr m \Rightarrow map (λ spt.
(MatchAnd (Match (C spt)) rst)) (singletonize-L4Ports m)
)
| (-, -) \Rightarrow undefined

lemma normalize-positive-ports-step-nnf:

assumes n: normalized-nnf-match m **and** wf-disc-sel: wf-disc-sel (disc,sel) C
and noneg: \neg has-disc-negated disc False m
shows $m' \in \text{set } (\text{normalize-positive-ports-step } (disc,sel) C m) \Rightarrow \text{normalized-nnf-match } m'$
<proof>

lemma normalize-positive-ports-step-normalized-n-primitive:

assumes n: normalized-nnf-match m **and** wf-disc-sel: wf-disc-sel (disc,sel) C
and noneg: \neg has-disc-negated disc False m
shows $\forall m' \in \text{set } (\text{normalize-positive-ports-step } (disc,sel) C m).$
normalized-n-primitive (disc,sel) (λ ps. case ps of L4Ports - pts \Rightarrow length
pts \leq 1) m'
<proof>

definition normalize-positive-src-ports :: 'i::len common-primitive match-expr \Rightarrow
'i common-primitive match-expr list **where**

normalize-positive-src-ports = normalize-positive-ports-step (is-Src-Ports, src-ports-sel)
Src-Ports

definition normalize-positive-dst-ports :: 'i::len common-primitive match-expr \Rightarrow
'i common-primitive match-expr list **where**

normalize-positive-dst-ports = normalize-positive-ports-step (is-Dst-Ports, dst-ports-sel)
Dst-Ports

lemma noNeg-mapNegPos-helper: getNeg ls = [] \Rightarrow

map (Pos \circ C) (getPos ls) = NegPos-map C ls

<proof>

lemma normalize-positive-src-ports:

assumes generic: primitive-matcher-generic β

and n: normalized-nnf-match m

and noneg: \neg has-disc-negated is-Src-Ports False m

shows

match-list (β , α) (normalize-positive-src-ports m) a p \longleftrightarrow matches (β , α)

m a p

<proof>

lemma normalize-positive-dst-ports:

assumes generic: primitive-matcher-generic β

and n : *normalized-nnf-match* m
and *noneg*: \neg *has-disc-negated is-Dst-Ports False* m
shows *match-list* (β, α) (*normalize-positive-dst-ports* m) $a p \longleftrightarrow$ *matches* $(\beta,$
 $\alpha)$ $m a p$
 \langle *proof* \rangle

lemma *normalize-positive-src-ports-nnf*:
assumes n : *normalized-nnf-match* m
and *noneg*: \neg *has-disc-negated is-Src-Ports False* m
shows $m' \in \text{set}$ (*normalize-positive-src-ports* m) \implies *normalized-nnf-match* m'
 \langle *proof* \rangle

lemma *normalize-positive-dst-ports-nnf*:
assumes n : *normalized-nnf-match* m
and *noneg*: \neg *has-disc-negated is-Dst-Ports False* m
shows $m' \in \text{set}$ (*normalize-positive-dst-ports* m) \implies *normalized-nnf-match* m'
 \langle *proof* \rangle

lemma *normalize-positive-src-ports-normalized-n-primitive*:
assumes n : *normalized-nnf-match* m
and *noneg*: \neg *has-disc-negated is-Src-Ports False* m
shows $\forall m' \in \text{set}$ (*normalize-positive-src-ports* m). *normalized-src-ports* m'
 \langle *proof* \rangle

lemma *normalize-positive-dst-ports-normalized-n-primitive*:
assumes n : *normalized-nnf-match* m
and *noneg*: \neg *has-disc-negated is-Dst-Ports False* m
shows $\forall m' \in \text{set}$ (*normalize-positive-dst-ports* m). *normalized-dst-ports* m'
 \langle *proof* \rangle

27.5 Complete Normalization

definition *normalize-ports-generic*
 $:: ('i \text{ common-primitive match-expr} \Rightarrow 'i \text{ common-primitive match-expr list}) \Rightarrow$
 $('i \text{ common-primitive match-expr} \Rightarrow 'i \text{ common-primitive match-expr}) \Rightarrow$
 $'i::\text{len common-primitive match-expr} \Rightarrow 'i \text{ common-primitive match-expr list}$

where

$\text{normalize-ports-generic normalize-pos rewrite-neg } m = \text{concat } (\text{map normalize-pos } (\text{normalize-match } (\text{rewrite-neg } m)))$

lemma *normalize-ports-generic-nnf*:
assumes n : *normalized-nnf-match* m
and *inset*: $m' \in \text{set}$ (*normalize-ports-generic normalize-pos rewrite-neg* m)
and *noNeg*: \neg *has-disc-negated disc False* (*rewrite-neg* m)
and *normalize-nnf-pos*: $\bigwedge m m'$
 $\text{normalized-nnf-match } m \implies \neg \text{has-disc-negated disc False } m \implies$
 $m' \in \text{set } (\text{normalize-pos } m) \implies \text{normalized-nnf-match } m'$

shows *normalized-nnf-match* m'
 ⟨*proof*⟩

lemma *normalize-ports-generic*:

assumes n : *normalized-nnf-match* m

and *normalize-pos*: $\bigwedge m. \text{normalized-nnf-match } m \implies \neg \text{has-disc-negated disc False } m \implies$

$\text{match-list } \gamma (\text{normalize-pos } m) a p \longleftrightarrow \text{matches } \gamma m a p$

and *rewrite-neg*: $\bigwedge m. \text{normalized-nnf-match } m \implies$

$\text{matches } \gamma (\text{rewrite-neg } m) a p = \text{matches } \gamma m a p$

and *noNeg*: $\bigwedge m. \text{normalized-nnf-match } m \implies \neg \text{has-disc-negated disc False } (\text{rewrite-neg } m)$

shows

$\text{match-list } \gamma (\text{normalize-ports-generic } \text{normalize-pos } \text{rewrite-neg } m) a p \longleftrightarrow \text{matches } \gamma m a p$

⟨*proof*⟩

lemma *normalize-ports-generic-normalized-n-primitive*:

assumes n : *normalized-nnf-match* m **and** *wf-disc-sel*: *wf-disc-sel* (disc , sel) C

and *noNeg*: $\bigwedge m. \text{normalized-nnf-match } m \implies \neg \text{has-disc-negated disc False } (\text{rewrite-neg } m)$

and *normalize-nnf-pos*: $\bigwedge m m'.$

$\text{normalized-nnf-match } m \implies \neg \text{has-disc-negated disc False } m \implies$

$m' \in \text{set } (\text{normalize-pos } m) \implies \text{normalized-nnf-match } m'$

and *normalize-pos*: $\bigwedge m m'.$

$\text{normalized-nnf-match } m \implies \neg \text{has-disc-negated disc False } m \implies$

$\forall m' \in \text{set } (\text{normalize-pos } m).$

$\text{normalized-n-primitive } (\text{disc}, \text{sel}) (\lambda ps. \text{case } ps \text{ of } L4Ports - pts \Rightarrow \text{length } pts \leq 1) m'$

shows $\forall m' \in \text{set } (\text{normalize-ports-generic } \text{normalize-pos } \text{rewrite-neg } m).$

$\text{normalized-n-primitive } (\text{disc}, \text{sel}) (\lambda ps. \text{case } ps \text{ of } L4Ports - pts \Rightarrow \text{length } pts \leq 1) m'$

⟨*proof*⟩

lemma *normalize-ports-generic-normalize-positive-ports-step-erule*:

assumes n : *normalized-nnf-match* m

and *wf-disc-sel*: *wf-disc-sel* (disc , sel) C

and *noProt*: $\forall a. \neg \text{disc } (\text{Prot } a)$

and P : $P (\text{disc2}, \text{sel2}) m$

and $P1$: $\bigwedge a. \text{normalized-nnf-match } a \implies$

$a \in \text{set } (\text{normalize-match } (\text{rewrite-negated-primitives } (\text{disc}, \text{sel}) C \text{ l4-ports-negate-one } m)) \implies$

$P (\text{disc2}, \text{sel2}) a$

and $P2$: $\bigwedge a \text{ dpts } \text{rst}. \text{normalized-nnf-match } a \implies$

$\text{primitive-extractor } (\text{disc}, \text{sel}) a = (\text{dpts}, \text{rst}) \implies$

$\text{getNeg } \text{dpts} = [] \implies P (\text{disc2}, \text{sel2}) a \implies P (\text{disc2}, \text{sel2}) \text{rst}$

and $P3$: $\bigwedge a \text{ spt } \text{rst}. P (\text{disc2}, \text{sel2}) \text{rst} \implies P (\text{disc2}, \text{sel2}) (\text{MatchAnd } (\text{Match } (C \text{ spt})) \text{rst})$

shows $m' \in \text{set } (\text{normalize-ports-generic } (\text{normalize-positive-ports-step } (\text{disc}, \text{sel}) C) (\text{rewrite-negated-primitives } (\text{disc}, \text{sel}) C \text{ l}_4\text{-ports-negate-one}) m) \implies$
 $P (\text{disc2}, \text{sel2}) m'$
 ⟨proof⟩

lemma *normalize-ports-generic-preserves-normalized-n-primitive:*

assumes $n: \text{normalized-nnf-match } m$
and $\text{wf-disc-sel}: \text{wf-disc-sel } (\text{disc}, \text{sel}) C$
and $\text{noProt}: \forall a. \neg \text{disc } (\text{Prot } a)$
and $\text{disc2-noC}: \forall a. \neg \text{disc2 } (C a)$
and $\text{disc2-noProt}: (\forall a. \neg \text{disc2 } (\text{Prot } a)) \vee \neg \text{has-disc-negated disc False } m$
shows $m' \in \text{set } (\text{normalize-ports-generic } (\text{normalize-positive-ports-step } (\text{disc}, \text{sel}) C) (\text{rewrite-negated-primitives } (\text{disc}, \text{sel}) C \text{ l}_4\text{-ports-negate-one}) m) \implies$
 $\text{normalized-n-primitive } (\text{disc2}, \text{sel2}) f m \implies$
 $\text{normalized-n-primitive } (\text{disc2}, \text{sel2}) f m'$
thm *normalize-ports-generic-normalize-positive-ports-step-erule*
 ⟨proof⟩

lemma *normalize-ports-generic-preserves-normalized-not-has-disc:*

assumes $n: \text{normalized-nnf-match } m$ **and** $\text{nodisc}: \neg \text{has-disc disc2 } m$
and $\text{wf-disc-sel}: \text{wf-disc-sel } (\text{disc}, \text{sel}) C$
and $\text{noProt}: \forall a. \neg \text{disc } (\text{Prot } a)$
and $\text{disc2-noC}: \forall a. \neg \text{disc2 } (C a)$
and $\text{disc2-noProt}: (\forall a. \neg \text{disc2 } (\text{Prot } a)) \vee \neg \text{has-disc-negated disc False } m$
shows $m' \in \text{set } (\text{normalize-ports-generic } (\text{normalize-positive-ports-step } (\text{disc}, \text{sel}) C) (\text{rewrite-negated-primitives } (\text{disc}, \text{sel}) C \text{ l}_4\text{-ports-negate-one}) m) \implies$
 $\neg \text{has-disc disc2 } m'$
 ⟨proof⟩

lemma *normalize-ports-generic-preserves-normalized-not-has-disc-negated:*

assumes $n: \text{normalized-nnf-match } m$ **and** $\text{nodisc}: \neg \text{has-disc-negated disc2 False } m$
and $\text{wf-disc-sel}: \text{wf-disc-sel } (\text{disc}, \text{sel}) C$
and $\text{noProt}: \forall a. \neg \text{disc } (\text{Prot } a)$
and $\text{disc2-noProt}: (\forall a. \neg \text{disc2 } (\text{Prot } a)) \vee \neg \text{has-disc-negated disc False } m$
shows $m' \in \text{set } (\text{normalize-ports-generic } (\text{normalize-positive-ports-step } (\text{disc}, \text{sel}) C) (\text{rewrite-negated-primitives } (\text{disc}, \text{sel}) C \text{ l}_4\text{-ports-negate-one}) m) \implies$
 $\neg \text{has-disc-negated disc2 False } m'$
 ⟨proof⟩

definition *normalize-src-ports*

$:: 'i::\text{len common-primitive match-expr} \Rightarrow 'i \text{ common-primitive match-expr list}$

where

$\text{normalize-src-ports } m = \text{normalize-ports-generic } \text{normalize-positive-src-ports } \text{rewrite-negated-src-ports } m$

definition *normalize-dst-ports*

$:: 'i::\text{len common-primitive match-expr} \Rightarrow 'i \text{ common-primitive match-expr list}$

where

$normalize_dst_ports\ m = normalize_ports_generic\ normalize_positive_dst_ports$
 $rewrite_negated_dst_ports\ m$

lemma *normalize-src-ports:*

assumes *generic: primitive-matcher-generic* β

and *n: normalized-nnf-match* m

shows *match-list* (β, α) $(normalize_src_ports\ m)$ $a\ p \longleftrightarrow matches\ (\beta, \alpha)\ m\ a\ p$
 $\langle proof \rangle$

lemma *normalize-dst-ports:*

assumes *generic: primitive-matcher-generic* β

and *n: normalized-nnf-match* m

shows *match-list* (β, α) $(normalize_dst_ports\ m)$ $a\ p \longleftrightarrow matches\ (\beta, \alpha)\ m\ a\ p$
 $\langle proof \rangle$

lemma *normalize-src-ports-normalized-n-primitive:*

assumes *n: normalized-nnf-match* m

shows $\forall m' \in set\ (normalize_src_ports\ m). normalized_src_ports\ m'$
 $\langle proof \rangle$

lemma *normalize-dst-ports-normalized-n-primitive:*

assumes *n: normalized-nnf-match* m

shows $\forall m' \in set\ (normalize_dst_ports\ m). normalized_dst_ports\ m'$
 $\langle proof \rangle$

lemma *normalize-src-ports-nnf:*

assumes *n: normalized-nnf-match* m

shows $m' \in set\ (normalize_src_ports\ m) \implies normalized_nnf_match\ m'$
 $\langle proof \rangle$

lemma *normalize-dst-ports-nnf:*

assumes *n: normalized-nnf-match* m

shows $m' \in set\ (normalize_dst_ports\ m) \implies normalized_nnf_match\ m'$
 $\langle proof \rangle$

lemma *normalize-src-ports-preserves-normalized-n-primitive:*

assumes *n: normalized-nnf-match* m

and *disc2-noC: $\forall a. \neg disc2\ (Src\ Ports\ a)$*

and *disc2-noProt: $(\forall a. \neg disc2\ (Prot\ a)) \vee \neg has_disc_negated\ is_Src\ Ports$*

False m

shows $m' \in set\ (normalize_src_ports\ m) \implies$
 $normalized_n_primitive\ (disc2, sel2)\ f\ m \implies$
 $normalized_n_primitive\ (disc2, sel2)\ f\ m'$
 $\langle proof \rangle$

lemma *normalize-dst-ports-preserves-normalized-n-primitive:*

assumes *n: normalized-nnf-match* m

and *disc2-noC*: $\forall a. \neg \text{disc2 } (\text{Dst-Ports } a)$
and *disc2-noProt*: $(\forall a. \neg \text{disc2 } (\text{Prot } a)) \vee \neg \text{has-disc-negated is-Dst-Ports}$
False m
shows $m' \in \text{set } (\text{normalize-dst-ports } m) \implies$
 $\text{normalized-n-primitive } (\text{disc2}, \text{sel2}) f m \implies$
 $\text{normalized-n-primitive } (\text{disc2}, \text{sel2}) f m'$
 ⟨proof⟩

lemma *normalize-src-ports-preserves-normalized-not-has-disc*:
assumes *n*: *normalized-nnf-match m* **and** *nodisc*: $\neg \text{has-disc disc2 } m$
and *disc2-noC*: $\forall a. \neg \text{disc2 } (\text{Src-Ports } a)$
and *disc2-noProt*: $(\forall a. \neg \text{disc2 } (\text{Prot } a)) \vee \neg \text{has-disc-negated is-Src-Ports}$
False m
shows $m' \in \text{set } (\text{normalize-src-ports } m)$
 $\implies \neg \text{has-disc disc2 } m'$
 ⟨proof⟩

lemma *normalize-dst-ports-preserves-normalized-not-has-disc*:
assumes *n*: *normalized-nnf-match m* **and** *nodisc*: $\neg \text{has-disc disc2 } m$
and *disc2-noC*: $\forall a. \neg \text{disc2 } (\text{Dst-Ports } a)$
and *disc2-noProt*: $(\forall a. \neg \text{disc2 } (\text{Prot } a)) \vee \neg \text{has-disc-negated is-Dst-Ports}$
False m
shows $m' \in \text{set } (\text{normalize-dst-ports } m)$
 $\implies \neg \text{has-disc disc2 } m'$
 ⟨proof⟩

lemma *normalize-src-ports-preserves-normalized-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m* **and** *nodisc*: $\neg \text{has-disc-negated disc2 } \text{False}$
m
and *disc2-noProt*: $(\forall a. \neg \text{disc2 } (\text{Prot } a)) \vee \neg \text{has-disc-negated is-Src-Ports}$
False m
shows $m' \in \text{set } (\text{normalize-src-ports } m)$
 $\implies \neg \text{has-disc-negated disc2 } \text{False } m'$
 ⟨proof⟩

lemma *normalize-dst-ports-preserves-normalized-not-has-disc-negated*:
assumes *n*: *normalized-nnf-match m* **and** *nodisc*: $\neg \text{has-disc-negated disc2 } \text{False}$
m
and *disc2-noProt*: $(\forall a. \neg \text{disc2 } (\text{Prot } a)) \vee \neg \text{has-disc-negated is-Dst-Ports}$
False m
shows $m' \in \text{set } (\text{normalize-dst-ports } m)$
 $\implies \neg \text{has-disc-negated disc2 } \text{False } m'$
 ⟨proof⟩

value[code] *normalize-src-ports*
 $(\text{MatchAnd } (\text{Match } (\text{Dst } (\text{IpAddrNetmask } (\text{ipv4addr-of-dotdecimal } (127, 0, 0, 0)) 8)))$
 $(\text{MatchAnd } (\text{Match } (\text{Prot } (\text{Proto } \text{TCP})))$
 $(\text{MatchNot } (\text{Match } (\text{Src-Ports } (\text{L4Ports } \text{UDP } [(80, 80)]))))))$

)

lemma *map opt-MatchAny-match-expr (normalize-src-ports*
 (MatchAnd (Match (Dst (IpAddrNetmask (ipv4addr-of-dotdecimal
(127, 0, 0, 0)) 8)))
 (MatchAnd (Match (Prot (Proto TCP)))
 (MatchNot (Match (Src-Ports (L4Ports UDP [(80,80)]))))
))) =
 [MatchAnd (MatchNot (Match (Prot (Proto UDP))) (MatchAnd (Match (Dst
(IpAddrNetmask 0x7F000000 8))) (Match (Prot (Proto TCP))))],
 MatchAnd (Match (Src-Ports (L4Ports UDP [(0, 79)])) (MatchAnd (Match (Dst
(IpAddrNetmask 0x7F000000 8))) (Match (Prot (Proto TCP))))],
 MatchAnd (Match (Src-Ports (L4Ports UDP [(81, 0xFFFF)])) (MatchAnd (Match
(Dst (IpAddrNetmask 0x7F000000 8))) (Match (Prot (Proto TCP)))))] <proof>

lemma *map opt-MatchAny-match-expr (normalize-src-ports*
 (MatchAnd (Match (Dst (IpAddrNetmask (ipv4addr-of-dotdecimal
(127, 0, 0, 0)) 8)))
 (MatchAnd (Match (Prot (Proto ICMP)))
 (MatchAnd (Match (Src-Ports (L4Ports TCP [(22,22)]))
 (MatchNot (Match (Src-Ports (L4Ports UDP [(80,80)]))))
)))
 =
 [MatchAnd (Match (Src-Ports (L4Ports TCP [(22, 22)]))
 (MatchAnd (MatchNot (Match (Prot (Proto UDP))) (MatchAnd (Match (Dst
(IpAddrNetmask 0x7F000000 8))) (Match (Prot (Proto ICMP)))))] <proof>

lemma *map opt-MatchAny-match-expr (normalize-src-ports*
 (MatchAnd (Match ((Src-Ports (L4Ports UDP [(21,21), (22,22)])) ::
32 common-primitive))
 (Match (Prot (Proto UDP))))
 =
 [MatchAnd (Match (Src-Ports (L4Ports UDP [(21, 22)])) (Match (Prot (Proto
UDP))))] <proof>

lemma *normalize-match (andfold-MatchExp (map (l4-ports-negate-one C) [])) =*
[MatchAny] <proof>

definition *replace-primitive-matchexpr*
 :: (('a ⇒ bool) × ('a ⇒ 'b)) ⇒ — disc-sel
 ('b negation-type ⇒ 'a match-expr) ⇒ — replace function

```

'a match-expr ⇒ 'a match-expr where
replace-primitive-matchexpr disc-sel replace-f m ≡
  let (as, rst) = primitive-extractor disc-sel m
  in if as = [] then m else
    MatchAnd
      (andfold-MatchExp (map replace-f as))
      rst

```

It does nothing of there is not even a primitive in it

```

lemma replace-primitive-matchexpr-unchanged-if-not-has-disc:
assumes n: normalized-nnf-match m
and wf-disc-sel: wf-disc-sel (disc,sel) C
and noDisc: ¬ has-disc disc m
shows replace-primitive-matchexpr (disc,sel) replace-f m = m
  ⟨proof⟩

```

```

lemma replace-primitive-matchexpr:
assumes n: normalized-nnf-match m and wf-disc-sel: wf-disc-sel disc-sel C
and replace-f: ∀ pt. matches γ (replace-f pt) a p ⟷
  matches γ (negation-type-to-match-expr-f C pt) a p
shows matches γ (replace-primitive-matchexpr disc-sel replace-f m) a p ⟷
matches γ m a p
  ⟨proof⟩

```

```

lemma replace-primitive-matchexpr-replaces-disc:
assumes n: normalized-nnf-match m and wf-disc-sel: wf-disc-sel (disc, sel) C
and replace-f: ∀ a. ¬ has-disc disc (replace-f a)
shows ¬ has-disc disc (replace-primitive-matchexpr (disc, sel) replace-f m)
  ⟨proof⟩

```

```

lemma replace-primitive-matchexpr-preserves-not-has-disc:
assumes n: normalized-nnf-match m and wf-disc-sel: wf-disc-sel (disc,sel) C
and nodisc: ¬ has-disc disc2 m
and replace-f: has-disc disc m ⟹ ∀ pts. ¬ has-disc disc2 (replace-f pts)
shows ¬ has-disc disc2 (replace-primitive-matchexpr (disc,sel) replace-f m)
  ⟨proof⟩

```

```

lemma normalize-replace-primitive-matchexpr-preserves-normalized-n-primitive:
assumes n: normalized-nnf-match m
and wf-disc-sel: wf-disc-sel (disc, sel) C
and replace-f:
  ∧ a m'. m' ∈ set (normalize-match (replace-f a)) ⟹ normalized-n-primitive
(disc2, sel2) f m'
and nprim: normalized-n-primitive (disc2, sel2) f m
and m': m' ∈ set (normalize-match (replace-primitive-matchexpr (disc,sel)
replace-f m))

```


shows *normalized-n-primitive* (*disc2*, *sel2*) *f m'*
 ⟨*proof*⟩

lemma *normalize-replace-primitive-matchexpr-preserves-normalized-not-has-disc:*

assumes *n*: *normalized-nnf-match m*
and *wf-disc-sel*: *wf-disc-sel (disc, sel) C*
and *nodisc*: \neg *has-disc disc2 m*
and *replace-f*: $\bigwedge a. \neg$ *has-disc disc2 (replace-f a)*
shows $m' \in \text{set } (\text{normalize-match } (\text{replace-primitive-matchexpr } (\text{disc,sel}) \text{ replace-f } m))$
 $\implies \neg$ *has-disc disc2 m'*
 ⟨*proof*⟩

lemma *normalize-replace-primitive-matchexpr-preserves-normalized-not-has-disc-negated:*

assumes *n*: *normalized-nnf-match m*
and *wf-disc-sel*: *wf-disc-sel (disc, sel) C*
and *nodisc*: \neg *has-disc-negated disc2 neg m*
and *replace-f*: $\bigwedge a. \neg$ *has-disc-negated disc2 neg (replace-f a)*
shows $m' \in \text{set } (\text{normalize-match } (\text{replace-primitive-matchexpr } (\text{disc,sel}) \text{ replace-f } m))$
 $\implies \neg$ *has-disc-negated disc2 neg m'*
 ⟨*proof*⟩

corollary *normalize-replace-primitive-matchexpr:*

assumes *n*: *normalized-nnf-match m*
and *replace-f*:
 $\bigwedge m. \text{normalized-nnf-match } m \implies$
 $\text{matches } \gamma (\text{replace-primitive-matchexpr } \text{disc-sel } \text{replace-f } m) a p \longleftrightarrow \text{matches}$
 $\gamma m a p$
shows
 $\text{match-list } \gamma (\text{normalize-match } (\text{replace-primitive-matchexpr } \text{disc-sel } \text{replace-f}$
 $m)) a p \longleftrightarrow$
 $\text{matches } \gamma m a p$
 ⟨*proof*⟩

fun *rewrite-MultiportPorts-one*

$:: \text{ipt-l4-ports negation-type} \Rightarrow 'i::\text{len common-primitive match-expr}$ **where**
 $\text{rewrite-MultiportPorts-one } (\text{Pos } \text{pts}) =$
 $\text{MatchOr } (\text{Match } (\text{Src-Ports } \text{pts})) (\text{Match } (\text{Dst-Ports } \text{pts})) |$
 $\text{rewrite-MultiportPorts-one } (\text{Neg } \text{pts}) =$
 $\text{MatchAnd } (\text{MatchNot } (\text{Match } (\text{Src-Ports } \text{pts}))) (\text{MatchNot } (\text{Match } (\text{Dst-Ports}$
 $\text{pts})))$

lemma *rewrite-MultiportPorts-one:*

assumes *generic*: *primitive-matcher-generic β* **and** *n*: *normalized-nnf-match m*
shows
 $\text{matches } (\beta, \alpha) (\text{replace-primitive-matchexpr } (\text{is-MultiportPorts}, \text{multiportports-sel})$
 $\text{rewrite-MultiportPorts-one } m) a p \longleftrightarrow$

matches (β, α) *m a p*
 ⟨*proof*⟩

lemma $\forall a. \neg \text{disc} (\text{Src-Ports } a) \implies \forall a. \neg \text{disc} (\text{Dst-Ports } a) \implies$
 $\text{normalized-n-primitive} (\text{disc}, \text{sel}) f m \implies$
 $\forall m' \in \text{set} (\text{normalize-match} (\text{rewrite-MultiportPorts-one } a)).$
 $\text{normalized-n-primitive} (\text{disc}, \text{sel}) f m'$
 ⟨*proof*⟩

lemma *rewrite-MultiportPorts-one-nodisc:*
 $\forall a. \neg \text{disc} (\text{Src-Ports } a) \implies \forall a. \neg \text{disc} (\text{Dst-Ports } a) \implies$
 $\neg \text{has-disc disc} (\text{rewrite-MultiportPorts-one } a)$
 $\forall a. \neg \text{disc} (\text{Src-Ports } a) \implies \forall a. \neg \text{disc} (\text{Dst-Ports } a) \implies$
 $\neg \text{has-disc-negated disc neg} (\text{rewrite-MultiportPorts-one } a)$
 ⟨*proof*⟩

definition *rewrite-MultiportPorts*
 $:: 'i::\text{len common-primitive match-expr} \Rightarrow 'i \text{ common-primitive match-expr list}$
where
 $\text{rewrite-MultiportPorts } m \equiv \text{normalize-match}$
 $(\text{replace-primitive-matchexpr} (\text{is-MultiportPorts}, \text{multiportports-sel}) \text{rewrite-MultiportPorts-one}$
 $m)$

lemma *rewrite-MultiportPorts:*
assumes *generic: primitive-matcher-generic* β
and *n: normalized-nnf-match* m
shows
 $\text{match-list} (\beta, \alpha) (\text{rewrite-MultiportPorts } m) a p \longleftrightarrow \text{matches} (\beta, \alpha) m a p$
 ⟨*proof*⟩

lemma *rewrite-MultiportPorts-normalized-nnf-match:*
 $m' \in \text{set} (\text{rewrite-MultiportPorts } m) \implies \text{normalized-nnf-match } m'$
 ⟨*proof*⟩

It does nothing of there is not even the primitive in it

lemma *rewrite-MultiportPorts-unchanged-if-not-has-disc:*
assumes *n: normalized-nnf-match* m
and *noDisc:* $\neg \text{has-disc is-MultiportPorts } m$
shows $\text{rewrite-MultiportPorts } m = [m]$
 ⟨*proof*⟩

lemma *rewrite-MultiportPorts-preserves-normalized-n-primitive:*
assumes *n: normalized-nnf-match* m
and *disc2-noSrcPorts:* $\forall a. \neg \text{disc2} (\text{Src-Ports } a)$
and *disc2-noDstPorts:* $\forall a. \neg \text{disc2} (\text{Dst-Ports } a)$
shows $m' \in \text{set} (\text{rewrite-MultiportPorts } m) \implies$
 $\text{normalized-n-primitive} (\text{disc2}, \text{sel2}) f m \implies$

normalized-n-primitive (disc2, sel2) f m'
 ⟨proof⟩

lemma *rewrite-MultiportPorts-preserves-normalized-not-has-disc:*

assumes *n: normalized-nnf-match m*
and *nodisc: ¬ has-disc disc2 m*
and *disc2-noSrcPorts: ∀ a. ¬ disc2 (Src-Ports a)*
and *disc2-noDstPorts: ∀ a. ¬ disc2 (Dst-Ports a)*
shows *m' ∈ set (rewrite-MultiportPorts m)*
 $\implies \neg \text{has-disc disc2 } m'$

⟨proof⟩

lemma *rewrite-MultiportPorts-preserves-normalized-not-has-disc-negated:*

assumes *n: normalized-nnf-match m*
and *nodisc: ¬ has-disc-negated disc2 neg m*
and *disc2-noSrcPorts: ∀ a. ¬ disc2 (Src-Ports a)*
and *disc2-noDstPorts: ∀ a. ¬ disc2 (Dst-Ports a)*
shows *m' ∈ set (rewrite-MultiportPorts m)*
 $\implies \neg \text{has-disc-negated disc2 neg } m'$

⟨proof⟩

lemma *rewrite-MultiportPorts-removes-MultiportsPorts:*

assumes *n: normalized-nnf-match m*
shows *m' ∈ set (rewrite-MultiportPorts m) $\implies \neg \text{has-disc is-MultiportPorts } m'$*

⟨proof⟩

end

theory *IpAddresses-Normalize*

imports *Common-Primitive-Lemmas*

begin

27.6 Normalizing IP Addresses

fun *normalized-src-ips :: 'i::len common-primitive match-expr \Rightarrow bool* **where**
normalized-src-ips MatchAny = True |
normalized-src-ips (Match (Src (IpAddrRange - -))) = False |
normalized-src-ips (Match (Src (IpAddr -))) = False |
normalized-src-ips (Match (Src (IpAddrNetmask - -))) = True |
normalized-src-ips (Match -) = True |
normalized-src-ips (MatchNot (Match (Src -))) = False |
normalized-src-ips (MatchNot (Match -)) = True |
normalized-src-ips (MatchAnd m1 m2) = (normalized-src-ips m1 \wedge normal-
ized-src-ips m2) |
normalized-src-ips (MatchNot (MatchAnd - -)) = False |
normalized-src-ips (MatchNot (MatchNot -)) = False |
normalized-src-ips (MatchNot (MatchAny)) = True

lemma *normalized-src-ips-def2*: *normalized-src-ips ms = normalized-n-primitive (is-Src, src-sel) normalized-cidr-ip ms*
 ⟨proof⟩

fun *normalized-dst-ips* :: 'i::len *common-primitive match-expr* ⇒ *bool* **where**
normalized-dst-ips MatchAny = True |
normalized-dst-ips (Match (Dst (IpAddrRange - -))) = False |
normalized-dst-ips (Match (Dst (IpAddr -))) = False |
normalized-dst-ips (Match (Dst (IpAddrNetmask - -))) = True |
normalized-dst-ips (Match -) = True |
normalized-dst-ips (MatchNot (Match (Dst -))) = False |
normalized-dst-ips (MatchNot (Match -)) = True |
normalized-dst-ips (MatchAnd m1 m2) = (normalized-dst-ips m1 ∧ normal-
ized-dst-ips m2) |
normalized-dst-ips (MatchNot (MatchAnd - -)) = False |
normalized-dst-ips (MatchNot (MatchNot -)) = False |
normalized-dst-ips (MatchNot MatchAny) = True

lemma *normalized-dst-ips-def2*: *normalized-dst-ips ms = normalized-n-primitive (is-Dst, dst-sel) normalized-cidr-ip ms*
 ⟨proof⟩

value *normalize-primitive-extract (is-Src, src-sel) Src ipt-iprange-compress*
 (*MatchAnd (MatchNot (Match ((Src-Ports (L4Ports TCP [(1,2)])):: 32 com-*
mon-primitive))) (Match (Src-Ports (L4Ports TCP [(1,2)])))))
value *normalize-primitive-extract (is-Src, src-sel) Src ipt-iprange-compress*
 (*MatchAnd (MatchNot (Match (Src (IpAddrNetmask (10::ipv4addr) 2)))*
 (*Match (Src-Ports (L4Ports TCP [(1,2)]))))*)
value *normalize-primitive-extract (is-Src, src-sel) Src ipt-iprange-compress*
 (*MatchAnd (Match (Src (IpAddrNetmask (10::ipv4addr) 2))) (MatchAnd*
 (*Match (Src (IpAddrNetmask 10 8)) (Match (Src-Ports (L4Ports TCP [(1,2)]))))*)
value *normalize-primitive-extract (is-Src, src-sel) Src ipt-iprange-compress*
 (*MatchAnd (Match (Src (IpAddrNetmask (10::ipv4addr) 2))) (MatchAnd*
 (*Match (Src (IpAddrNetmask 192 8)) (Match (Src-Ports (L4Ports TCP [(1,2)]))))*)

definition *normalize-src-ips* :: 'i::len *common-primitive match-expr* ⇒ 'i *com-*
mon-primitive match-expr list **where**
normalize-src-ips = normalize-primitive-extract (common-primitive.is-Src, src-sel)
common-primitive.Src ipt-iprange-compress

lemma *ipt-iprange-compress-src-matching*: *match-list (common-matcher, α) (map*
 (*Match* ∘ *Src*) (*ipt-iprange-compress ml*)) *a p* ⇔
matches (common-matcher, α) (alist-and (NegPos-map Src ml)) a p
 ⟨proof⟩

lemma *normalize-src-ips*: *normalized-nnf-match m* ⇒

$match-list (common-matcher, \alpha) (normalize-src-ips m) a p = matches (common-matcher, \alpha) m a p$
 <proof>

lemma *normalize-src-ips-normalized-n-primitive: normalized-nnf-match m \implies*
 $\forall m' \in set (normalize-src-ips m). normalized-src-ips m'$
 <proof>

definition *normalize-dst-ips :: 'i::len common-primitive match-expr \Rightarrow 'i com-*
mon-primitive match-expr list where
normalize-dst-ips = normalize-primitive-extract (common-primitive.is-Dst, dst-sel)
common-primitive.Dst ipt-iprange-compress

lemma *ipt-iprange-compress-dst-matching: match-list (common-matcher, \alpha) (map*
(Match \circ Dst) (ipt-iprange-compress ml)) a p \longleftrightarrow
matches (common-matcher, \alpha) (alist-and (NegPos-map Dst ml)) a p
 <proof>

lemma *normalize-dst-ips: normalized-nnf-match m \implies*
match-list (common-matcher, \alpha) (normalize-dst-ips m) a p = matches (common-matcher,
\alpha) m a p
 <proof>

Normalizing the dst ips preserves the normalized src ips

lemma *normalized-nnf-match m \implies normalized-src-ips m $\implies \forall mn \in set (normalize-dst-ips$*
m). normalized-src-ips mn
 <proof>

lemma *normalize-dst-ips-normalized-n-primitive: normalized-nnf-match m \implies*
 $\forall m' \in set (normalize-dst-ips m). normalized-dst-ips m'$
 <proof>

end

theory *Interfaces-Normalize*

imports *Common-Primitive-Lemmas*

begin

27.7 Optimizing interfaces in match expressions

definition *compress-interfaces :: iface negation-type list \Rightarrow (iface list \times iface list)*
option where

compress-interfaces ifces \equiv case (compress-pos-interfaces (getPos ifces))
of None \Rightarrow None
| Some i \Rightarrow if
$\exists negated-ifce \in set (getNeg ifces). iface-subset i negated-ifce$
then
None

```

else if
  ¬ iface-is-wildcard i
then
  Some ([i], [])
else
  Some ((if i = ifaceAny then [] else [i]), getNeg ifces)

```

context

begin

private lemma *compress-interfaces-None:*

assumes *generic: primitive-matcher-generic* β

shows

compress-interfaces ifces = None \implies \neg *matches* (β , α) (*alist-and* (*NegPos-map* *IIface ifces*)) *a p*

compress-interfaces ifces = None \implies \neg *matches* (β , α) (*alist-and* (*NegPos-map* *OIface ifces*)) *a p*

<proof> **lemma** *compress-interfaces-Some:*

assumes *generic: primitive-matcher-generic* β

shows

compress-interfaces ifces = Some (i-pos, i-neg) \implies

matches (β , α) (*alist-and* (*NegPos-map* *IIface* ((*map* *Pos i-pos*)@(map *Neg* *i-neg*)))) *a p* \longleftrightarrow

matches (β , α) (*alist-and* (*NegPos-map* *IIface ifces*)) *a p*

compress-interfaces ifces = Some (i-pos, i-neg) \implies

matches (β , α) (*alist-and* (*NegPos-map* *OIface* ((*map* *Pos i-pos*)@(map *Neg* *i-neg*)))) *a p* \longleftrightarrow

matches (β , α) (*alist-and* (*NegPos-map* *OIface ifces*)) *a p*

<proof>

definition *compress-normalize-input-interfaces* :: *'i::len common-primitive match-expr*
 \Rightarrow *'i common-primitive match-expr option* **where**

compress-normalize-input-interfaces m \equiv *compress-normalize-primitive (is-Iiface, iiface-sel) IIface compress-interfaces m*

lemma *compress-normalize-input-interfaces-Some:*

assumes *generic: primitive-matcher-generic* β

and *normalized-nnf-match m* **and** *compress-normalize-input-interfaces m = Some m'*

shows *matches* (β , α) *m' a p* \longleftrightarrow *matches* (β , α) *m a p*

<proof>

lemma *compress-normalize-input-interfaces-None:*

assumes *generic: primitive-matcher-generic* β

and *normalized-nnf-match m* **and** *compress-normalize-input-interfaces m = None*

shows \neg *matches* (β , α) *m a p*

<proof>

lemma *compress-normalize-input-interfaces-nnf*: *normalized-nnf-match* $m \implies$
compress-normalize-input-interfaces $m = \text{Some } m' \implies$
normalized-nnf-match m'
 ⟨proof⟩

lemma *compress-normalize-input-interfaces-not-introduces-Iiface*:
 $\neg \text{has-disc is-Iiface } m \implies \text{normalized-nnf-match } m \implies \text{compress-normalize-input-interfaces}$
 $m = \text{Some } m' \implies$
 $\neg \text{has-disc is-Iiface } m'$
 ⟨proof⟩

lemma *compress-normalize-input-interfaces-not-introduces-Iiface-negated*:
assumes *notdisc*: $\neg \text{has-disc-negated is-Iiface False } m$
and *nm*: *normalized-nnf-match* m
and *some*: *compress-normalize-input-interfaces* $m = \text{Some } m'$
shows $\neg \text{has-disc-negated is-Iiface False } m'$
 ⟨proof⟩

lemma *compress-normalize-input-interfaces-hasdisc*:
 $\neg \text{has-disc disc } m \implies (\forall a. \neg \text{disc (Iiface } a)) \implies \text{normalized-nnf-match } m \implies$
compress-normalize-input-interfaces $m = \text{Some } m' \implies$
normalized-nnf-match $m' \wedge \neg \text{has-disc disc } m'$
 ⟨proof⟩

lemma *compress-normalize-input-interfaces-hasdisc-negated*:
 $\neg \text{has-disc-negated disc neg } m \implies (\forall a. \neg \text{disc (Iiface } a)) \implies \text{normalized-nnf-match}$
 $m \implies \text{compress-normalize-input-interfaces } m = \text{Some } m' \implies$
normalized-nnf-match $m' \wedge \neg \text{has-disc-negated disc neg } m'$
 ⟨proof⟩

lemma *compress-normalize-input-interfaces-preserves-normalized-n-primitive*:
normalized-n-primitive (*disc*, *sel*) $P m \implies (\forall a. \neg \text{disc (Iiface } a)) \implies \text{normal-}$
ized-nnf-match $m \implies \text{compress-normalize-input-interfaces } m = \text{Some } m' \implies$
normalized-nnf-match $m' \wedge \text{normalized-n-primitive (disc, sel) } P m'$
 ⟨proof⟩

value[code] *compress-normalize-input-interfaces*
 (*MatchAnd* (*MatchAnd* (*MatchAnd* (*Match* ((*Iiface* (*Iface* "eth+"))::32 *com-*
mon-primitive))) (*MatchNot* (*Match* (*Iiface* (*Iface* "eth4"))))) (*Match* (*Iiface* (*Iface*
 "eth1"))))
 (*Match* (*Prot* (*Proto* *TCP*))))

value[code] *compress-normalize-input-interfaces* (*MatchAny*:: 32 *common-primitive*

match-expr)

definition *compress-normalize-output-interfaces* :: 'i::len *common-primitive match-expr*
⇒ 'i *common-primitive match-expr option* **where**
 compress-normalize-output-interfaces m ≡ *compress-normalize-primitive* (*is-Oiface*,
oiface-sel) *Oiface* *compress-interfaces* m

lemma *compress-normalize-output-interfaces-Some*:
assumes *generic: primitive-matcher-generic* β
 and *normalized-nnf-match* m **and** *compress-normalize-output-interfaces* m =
Some m'
shows *matches* (β, α) m' a p ↔ *matches* (β, α) m a p
 ⟨*proof*⟩

lemma *compress-normalize-output-interfaces-None*:
assumes *generic: primitive-matcher-generic* β
 and *normalized-nnf-match* m **and** *compress-normalize-output-interfaces* m =
None
shows ¬ *matches* (β, α) m a p
 ⟨*proof*⟩

lemma *compress-normalize-output-interfaces-nnf*: *normalized-nnf-match* m ⇒
compress-normalize-output-interfaces m = *Some* m' ⇒
 normalized-nnf-match m'
 ⟨*proof*⟩

lemma *compress-normalize-output-interfaces-not-introduces-Oiface*:
¬ *has-disc is-Oiface* m ⇒ *normalized-nnf-match* m ⇒ *compress-normalize-output-interfaces*
m = *Some* m' ⇒
 ¬ *has-disc is-Oiface* m'
 ⟨*proof*⟩

lemma *compress-normalize-output-interfaces-not-introduces-Oiface-negated*:
assumes *notdisc*: ¬ *has-disc-negated is-Oiface* *False* m
 and *nm*: *normalized-nnf-match* m
 and *some*: *compress-normalize-output-interfaces* m = *Some* m'
shows ¬ *has-disc-negated is-Oiface* *False* m'
 ⟨*proof*⟩

lemma *compress-normalize-output-interfaces-hasdisc*:
¬ *has-disc disc* m ⇒ (∀ a. ¬ *disc* (*Oiface* a)) ⇒ *normalized-nnf-match* m
⇒ *compress-normalize-output-interfaces* m = *Some* m' ⇒
 normalized-nnf-match m' ∧ ¬ *has-disc disc* m'
 ⟨*proof*⟩

lemma *compress-normalize-output-interfaces-hasdisc-negated:*
 $\neg \text{has-disc-negated } \text{disc } \text{neg } m \implies (\forall a. \neg \text{disc } (\text{OIface } a)) \implies \text{normalized-nnf-match } m \implies \text{compress-normalize-output-interfaces } m = \text{Some } m' \implies$
 $\text{normalized-nnf-match } m' \wedge \neg \text{has-disc-negated } \text{disc } \text{neg } m'$
 $\langle \text{proof} \rangle$

lemma *compress-normalize-output-interfaces-preserves-normalized-n-primitive:*
 $\text{normalized-n-primitive } (\text{disc}, \text{sel}) P m \implies (\forall a. \neg \text{disc } (\text{OIface } a)) \implies \text{normalized-nnf-match } m \implies \text{compress-normalize-output-interfaces } m = \text{Some } m' \implies$
 $\text{normalized-nnf-match } m' \wedge \text{normalized-n-primitive } (\text{disc}, \text{sel}) P m'$
 $\langle \text{proof} \rangle$

end

end

28 Word Upto

theory *Word-Upto*

imports *Main*

IP-Addresses.Hs-Compat

Word-Lib.Word-Lemmas

begin

Enumerate a range of machine words.

enumerate from the back (inefficient)

function *word-upto* :: 'a word \Rightarrow 'a word \Rightarrow ('a::len) word list **where**
word-upto a b = (if a = b then [a] else *word-upto* a (b - 1) @ [b])
 $\langle \text{proof} \rangle$

termination *word-upto*

$\langle \text{proof} \rangle$

declare *word-upto.simps*[*simp del*]

enumerate from the front (more inefficient)

function *word-upto'* :: 'a word \Rightarrow 'a word \Rightarrow ('a::len) word list **where**
word-upto' a b = (if a = b then [a] else a # *word-upto'* (a + 1) b)
 $\langle \text{proof} \rangle$

termination *word-upto'*

$\langle \text{proof} \rangle$

declare *word-upto'.simps*[*simp del*]

lemma *word-upto-cons-front*[code]:
word-upto a b = word-upto' a b
 ⟨*proof*⟩

lemma *word-upto-set-eq*: $a \leq b \implies x \in \text{set } (\text{word-upto } a \ b) \iff a \leq x \wedge x \leq b$
 ⟨*proof*⟩

lemma *word-upto-distinct-hlp*: $a \leq b \implies a \neq b \implies b \notin \text{set } (\text{word-upto } a \ (b - 1))$
 ⟨*proof*⟩

lemma *distinct-word-upto*: $a \leq b \implies \text{distinct } (\text{word-upto } a \ b)$
 ⟨*proof*⟩

lemma *word-upto-eq-upto*: $s \leq e \implies e \leq \text{unat } (\text{max-word} :: 'l \ \text{word}) \implies$
 $\text{word-upto } ((\text{of-nat} :: \text{nat} \Rightarrow ('l :: \text{len}) \ \text{word}) \ s) \ (\text{of-nat } e) = \text{map of-nat } (\text{upt}$
 $s \ (\text{Suc } e))$
 ⟨*proof*⟩

lemma *word-upto-alt*: $(a :: ('l :: \text{len}) \ \text{word}) \leq b \implies$
 $\text{word-upto } a \ b = \text{map of-nat } (\text{upt } (\text{unat } a) \ (\text{Suc } (\text{unat } b)))$
 ⟨*proof*⟩

lemma *word-upto-upt*:
 $\text{word-upto } a \ b = (\text{if } a \leq b \ \text{then } \text{map of-nat } (\text{upt } (\text{unat } a) \ (\text{Suc } (\text{unat } b)))) \ \text{else}$
 $\text{word-upto } a \ b)$
 ⟨*proof*⟩

lemma *sorted-word-upto*:
fixes $a \ b :: ('l :: \text{len}) \ \text{word}$
assumes $a \leq b$
shows *sorted* $(\text{word-upto } a \ b)$
 ⟨*proof*⟩

end
theory *Protocols-Normalize*
imports *Common-Primitive-Lemmas*
../Common/Word-Upto
begin

29 Optimizing Protocols

30 Optimizing protocols in match expressions

fun *compress-pos-protocols* :: *protocol list* \Rightarrow *protocol option* **where**
compress-pos-protocols [] = *Some ProtoAny* |
compress-pos-protocols [p] = *Some p* |
compress-pos-protocols (p1 # p2 # ps) = (case *simple-proto-conjunct* p1 p2 of
None \Rightarrow *None* | *Some p* \Rightarrow *compress-pos-protocols* (p # ps))

lemma *compress-pos-protocols-Some*: *compress-pos-protocols* ps = *Some proto*
 \implies
match-proto proto p-prot \longleftrightarrow (\forall p \in set ps. *match-proto p p-prot*)
 <proof>

lemma *compress-pos-protocols-None*: *compress-pos-protocols* ps = *None* \implies
 \neg (\forall proto \in set ps. *match-proto proto p-prot*)
 <proof>

lemma *simple-proto-conjunct* (*Proto p1*) (*Proto p2*) \neq *None* \implies \forall pkt. *match-proto*
 (*Proto p1*) pkt \longleftrightarrow *match-proto* (*Proto p2*) pkt
 <proof>

lemma *simple-proto-conjunct* p1 (*Proto p2*) \neq *None* \implies \forall pkt. *match-proto* (*Proto*
 p2) pkt \implies *match-proto* p1 pkt
 <proof>

definition *compress-protocols* :: *protocol negation-type list* \Rightarrow (*protocol list* \times
protocol list) *option* **where**
compress-protocols ps \equiv case (*compress-pos-protocols* (*getPos* ps))
 of *None* \Rightarrow *None*
 | *Some proto* \Rightarrow if *ProtoAny* \in set (*getNeg* ps) \vee (\forall p \in {0..- 1}. *Proto p*
 \in set (*getNeg* ps)) then

None
 else if *proto* = *ProtoAny* then
Some ([], *getNeg* ps)
 else if (\exists p \in set (*getNeg* ps). *simple-proto-conjunct proto p* \neq
None) then

None
 else
 — *proto* is a *primitive-protocol* here. This is strict equality
 match, e.g.
 — protocol must be TCP. Thus, we can remove all negative
 matches!
Some ([*proto*], [])

lemma *all-proto-hlp2*: *ProtoAny* \in a \vee (\forall p \in {0..- 1}. *Proto p* \in a) \longleftrightarrow
ProtoAny \in a \vee a = {p. p \neq *ProtoAny*}

<proof>

lemma *set-word8-word-upto*: $\{0..(- 1 :: 8 \text{ word})\} = \text{set} (\text{word-upto } 0 \ 255)$

<proof>

lemma $(\forall p \in \{0..- 1\}. \text{Proto } p \in \text{set} (\text{getNeg } ps)) \longleftrightarrow$
 $((\forall p \in \text{set} (\text{word-upto } 0 \ 255). \text{Proto } p \in \text{set} (\text{getNeg } ps)))$
<proof>

lemma *compress-protocols-code*[code]:

compress-protocols ps = (case (compress-pos-protocols (getPos ps))
of None \Rightarrow None
| Some proto \Rightarrow if ProtoAny \in set (getNeg ps) \vee ($\forall p \in$ set (word-upto 0
255). Proto p \in set (getNeg ps)) then
None
else if proto = ProtoAny then
Some ([], getNeg ps)
else if ($\exists p \in$ set (getNeg ps). simple-proto-conjunct proto p \neq
None) then
None
else
Some ([proto], [])
)
<proof>

lemma *compress-protocols ps = Some (ps-pos, ps-neg) \implies*
 $\exists p. ((\forall m \in \text{set } ps\text{-pos}. \text{match-proto } m \ p) \wedge (\forall m \in \text{set } ps\text{-neg}. \neg \text{match-proto } m$
p))
<proof>

definition *compress-normalize-protocols-step* :: 'i::len common-primitive match-expr
 \Rightarrow 'i common-primitive match-expr option **where**
compress-normalize-protocols-step m \equiv compress-normalize-primitive (is-Prot,
prot-sel) Prot compress-protocols m

lemma (in *primitive-matcher-generic*) *compress-normalize-protocols-step-Some:*
assumes *normalized-nnf-match m and compress-normalize-protocols-step m =*
Some m'
shows *matches (β , α) m' a p \longleftrightarrow matches (β , α) m a p*
<proof>

lemma (in *primitive-matcher-generic*) *compress-normalize-protocols-step-None:*
assumes *normalized-nnf-match m and compress-normalize-protocols-step m =*
None
shows $\neg \text{matches } (\beta, \alpha) \ m \ a \ p$
<proof>

lemma *compress-normalize-protocols-step-nnf:*

$normalized_nnf_match\ m \implies compress_normalize_protocols_step\ m = Some\ m'$
 \implies
 $normalized_nnf_match\ m'$
 $\langle proof \rangle$

lemma *compress-normalize-protocols-step-not-introduces-Prot*:
 $\neg has_disc\ is_Prot\ m \implies normalized_nnf_match\ m \implies compress_normalize_protocols_step\ m = Some\ m' \implies$
 $\neg has_disc\ is_Prot\ m'$
 $\langle proof \rangle$

lemma *compress-normalize-protocols-step-not-introduces-Prot-negated*:
assumes *notdisc*: $\neg has_disc_negated\ is_Prot\ False\ m$
and *nm*: $normalized_nnf_match\ m$
and *some*: $compress_normalize_protocols_step\ m = Some\ m'$
shows $\neg has_disc_negated\ is_Prot\ False\ m'$
 $\langle proof \rangle$

lemma *compress-normalize-protocols-step-hasdisc*:
 $\neg has_disc\ disc\ m \implies (\forall a. \neg disc\ (Prot\ a)) \implies normalized_nnf_match\ m \implies$
 $compress_normalize_protocols_step\ m = Some\ m' \implies$
 $normalized_nnf_match\ m' \wedge \neg has_disc\ disc\ m'$
 $\langle proof \rangle$

lemma *compress-normalize-protocols-step-hasdisc-negated*:
 $\neg has_disc_negated\ disc\ neg\ m \implies (\forall a. \neg disc\ (Prot\ a)) \implies normalized_nnf_match\ m \implies$
 $compress_normalize_protocols_step\ m = Some\ m' \implies$
 $normalized_nnf_match\ m' \wedge \neg has_disc_negated\ disc\ neg\ m'$
 $\langle proof \rangle$

lemma *compress-normalize-protocols-step-preserves-normalized-n-primitive*:
 $normalized_n_primitive\ (disc,\ sel)\ P\ m \implies (\forall a. \neg disc\ (Prot\ a)) \implies normalized_nnf_match\ m \implies$
 $compress_normalize_protocols_step\ m = Some\ m' \implies$
 $normalized_nnf_match\ m' \wedge normalized_n_primitive\ (disc,\ sel)\ P\ m'$
 $\langle proof \rangle$

lemma *case compress-normalize-protocols-step*
 $(MatchAnd\ (MatchAnd\ (MatchAnd\ (Match\ ((Prot\ (Proto\ TCP))::\ 32\ common_primitive)))\ (MatchNot\ (Match\ (Prot\ (Proto\ UDP)))))\ (Match\ (Iiface\ (Iface\ "eth1"))))$
 $(Match\ (Prot\ (Proto\ TCP)))\ of\ Some\ ps \Rightarrow opt_MatchAny_match_expr$
 ps
 $= MatchAnd\ (Match\ (Prot\ (Proto\ 6)))\ (Match\ (Iiface\ (Iface\ "eth1")))\ \langle proof \rangle$

value[code] *compress-normalize-protocols-step* $(MatchAny::\ 32\ common_primitive$

match-expr)

30.1 Importing the matches on *primitive-protocol* from *L4Ports*

definition *import-protocols-from-ports*

$:: 'i::len$ *common-primitive match-expr* $\Rightarrow 'i$ *common-primitive match-expr*

where

import-protocols-from-ports $m \equiv$
 (case *primitive-extractor* (*is-Src-Ports*, *src-ports-sel*) m of (*srcpts*, *rst1*) \Rightarrow
 case *primitive-extractor* (*is-Dst-Ports*, *dst-ports-sel*) *rst1* of (*dstpts*, *rst2*) \Rightarrow
MatchAnd
 (*MatchAnd*
 (*MatchAnd*
 (*andfold-MatchExp* (*map* (*Match* \circ (*Prot* \circ (*case-ipt-l4-ports* ($\lambda proto$ x . *Proto*
proto)))) (*getPos* *srcpts*)))
 (*andfold-MatchExp* (*map* (*Match* \circ (*Prot* \circ (*case-ipt-l4-ports* ($\lambda proto$ x . *Proto*
proto)))) (*getPos* *dstpts*)))
)
 (*alist-and'* (*NegPos-map Src-Ports* *srcpts* @ *NegPos-map Dst-Ports* *dstpts*))
)
rst2
)

The *Proto* and *L4Ports* match make the following match impossible:

lemma *compress-normalize-protocols-step* (*import-protocols-from-ports*

(*MatchAnd* (*MatchAnd* (*Match* (*Prot* (*Proto* *TCP*)):: 32 *common-primitive*))

(*Match* (*Src-Ports* (*L4Ports* *UDP* [(22,22)]))) (*Match* (*Iiface* (*Iface* "eth1"))))

= *None*

<proof>

lemma *import-protocols-from-ports-erule*: *normalized-nnf-match* $m \Longrightarrow P$ $m \Longrightarrow$

(\bigwedge *srcpts* *rst1* *dstpts* *rst2*.

normalized-nnf-match $m \Longrightarrow$

— P $m \Longrightarrow$ *erule* consumes only first argument

primitive-extractor (*is-Src-Ports*, *src-ports-sel*) $m =$ (*srcpts*, *rst1*) \Longrightarrow

primitive-extractor (*is-Dst-Ports*, *dst-ports-sel*) *rst1* = (*dstpts*, *rst2*) \Longrightarrow

normalized-nnf-match *rst1* \Longrightarrow

normalized-nnf-match *rst2* \Longrightarrow

P (*MatchAnd*

(*MatchAnd*

(*MatchAnd*

(*andfold-MatchExp*

(*map* (*Match* \circ (*Prot* \circ (*case-ipt-l4-ports* ($\lambda proto$ x . *Proto* *proto*))))

(*getPos* *srcpts*)))

(*andfold-MatchExp*

(*map* (*Match* \circ (*Prot* \circ (*case-ipt-l4-ports* ($\lambda proto$ x . *Proto* *proto*))))

(*getPos* *dstpts*)))

$(alist-and' (NegPos-map Src-Ports srcpts @ NegPos-map Dst-Ports dstpts)))$
 $rst2)) \implies$
 $P (import-protocols-from-ports m)$
 $\langle proof \rangle$

lemma (in *primitive-matcher-generic*) *import-protocols-from-ports*:
assumes *normalized: normalized-nnf-match m*
shows *matches* (β, α) (*import-protocols-from-ports m*) $a p \iff matches$ (β, α)
 $m a p$
 $\langle proof \rangle$

lemma *import-protocols-from-ports-nnf*:
 $normalized-nnf-match m \implies normalized-nnf-match (import-protocols-from-ports m)$
 $\langle proof \rangle$

lemma *import-protocols-from-ports-not-introduces-Prot-negated*:
 $normalized-nnf-match m \implies \neg has-disc-negated is-Prot False m \implies$
 $\neg has-disc-negated is-Prot False (import-protocols-from-ports m)$
 $\langle proof \rangle$

lemma *import-protocols-from-ports-hasdisc*:
 $normalized-nnf-match m \implies \neg has-disc disc m \implies (\forall a. \neg disc (Prot a)) \implies$
 $normalized-nnf-match (import-protocols-from-ports m) \wedge \neg has-disc disc (import-protocols-from-ports m)$
 $\langle proof \rangle$

lemma *import-protocols-from-ports-hasdisc-negated*:
 $\neg has-disc-negated disc False m \implies (\forall a. \neg disc (Prot a)) \implies normalized-nnf-match m \implies$
 $normalized-nnf-match (import-protocols-from-ports m) \wedge$
 $\neg has-disc-negated disc False (import-protocols-from-ports m)$
 $\langle proof \rangle$

lemma *import-protocols-from-ports-preserves-normalized-n-primitive*:
 $normalized-n-primitive (disc, sel) f m \implies (\forall a. \neg disc (Prot a)) \implies normalized-nnf-match m \implies$
 $normalized-nnf-match (import-protocols-from-ports m) \wedge normalized-n-primitive (disc, sel) f (import-protocols-from-ports m)$
 $\langle proof \rangle$

30.2 Putting things together

definition *compress-normalize-protocols*

$:: 'i::len \text{ common-primitive match-expr} \Rightarrow 'i \text{ common-primitive match-expr option}$ **where**

$\text{compress-normalize-protocols } m \equiv \text{compress-normalize-protocols-step (import-protocols-from-ports } m)$

lemma (*in primitive-matcher-generic*) *compress-normalize-protocols-Some:*

assumes *normalized-nnf-match* m **and** *compress-normalize-protocols* $m = \text{Some } m'$

shows $\text{matches } (\beta, \alpha) m' a p \longleftrightarrow \text{matches } (\beta, \alpha) m a p$
<proof>

lemma (*in primitive-matcher-generic*) *compress-normalize-protocols-None:*

assumes *normalized-nnf-match* m **and** *compress-normalize-protocols* $m = \text{None}$

shows $\neg \text{matches } (\beta, \alpha) m a p$
<proof>

lemma *compress-normalize-protocols-nnf:*

$\text{normalized-nnf-match } m \Longrightarrow \text{compress-normalize-protocols } m = \text{Some } m' \Longrightarrow$
 $\text{normalized-nnf-match } m'$

<proof>

lemma *compress-normalize-protocols-not-introduces-Prot-negated:*

assumes *notdisc:* $\neg \text{has-disc-negated is-Prot False } m$

and *nm:* *normalized-nnf-match* m

and *some:* *compress-normalize-protocols* $m = \text{Some } m'$

shows $\neg \text{has-disc-negated is-Prot False } m'$
<proof>

lemma *compress-normalize-protocols-hasdisc:*

$\neg \text{has-disc disc } m \Longrightarrow (\forall a. \neg \text{disc (Prot } a)) \Longrightarrow \text{normalized-nnf-match } m \Longrightarrow$
 $\text{compress-normalize-protocols } m = \text{Some } m' \Longrightarrow$
 $\text{normalized-nnf-match } m' \wedge \neg \text{has-disc disc } m'$

<proof>

lemma *compress-normalize-protocols-hasdisc-negated:*

$\neg \text{has-disc-negated disc False } m \Longrightarrow (\forall a. \neg \text{disc (Prot } a)) \Longrightarrow$

$\text{normalized-nnf-match } m \Longrightarrow \text{compress-normalize-protocols } m = \text{Some } m' \Longrightarrow$
 $\text{normalized-nnf-match } m' \wedge \neg \text{has-disc-negated disc False } m'$

<proof>

lemma *compress-normalize-protocols-preserves-normalized-n-primitive:*

$\text{normalized-n-primitive (disc, sel) } P m \Longrightarrow (\forall a. \neg \text{disc (Prot } a)) \Longrightarrow \text{normal-}$
 $\text{ized-nnf-match } m \Longrightarrow \text{compress-normalize-protocols } m = \text{Some } m' \Longrightarrow$
 $\text{normalized-nnf-match } m' \wedge \text{normalized-n-primitive (disc, sel) } P m'$

<proof>

lemma *case compress-normalize-protocols*

(MatchAnd (MatchAnd (MatchAnd (Match ((Prot (Proto TCP)):: 32 common-primitive)) (MatchNot (Match (Prot (Proto UDP)))))) (Match (Iiface (Iface "eth1"))))

(Match (Prot (Proto TCP))) of Some ps ⇒ opt-MatchAny-match-expr

ps

=

MatchAnd (Match (Prot (Proto 6))) (Match (Iiface (Iface "eth1")) <proof>

value_[code] *compress-normalize-protocols (MatchAny:: 32 common-primitive match-expr)*

end

31 Reverse Remdups

theory *Remdups-Rev*

imports *Main*

begin

definition *remdups-rev :: 'a list ⇒ 'a list where*

remdups-rev rs ≡ rev (remdups (rev rs))

lemma *remdups-append: remdups (rs @ rs2) = remdups [r←rs . r ∉ set rs2] @ remdups rs2*

<proof>

lemma *remdups-rev-append: remdups-rev (rs @ rs2) = remdups-rev rs @ remdups-rev [r←rs2 . r ∉ set rs]*

<proof>

lemma *remdups-rev-fst:*

remdups-rev (r#rs) = (if r ∈ set rs then r#remdups-rev (removeAll r rs) else r#remdups-rev rs)

<proof>

lemma *remdups-rev-set: set (remdups-rev rs) = set rs <proof>*

lemma *remdups-rev-removeAll: remdups-rev (removeAll r rs) = removeAll r (remdups-rev rs)*

<proof>

Faster code equations

fun *remdups-rev-code :: 'a list ⇒ 'a list ⇒ 'a list where*

remdups-rev-code - [] = [] |

$remdups\text{-}rev\text{-}code\ ps\ (r\#rs) = (if\ r \in set\ ps\ then\ remdups\text{-}rev\text{-}code\ ps\ rs\ else\ r\#remdups\text{-}rev\text{-}code\ (r\#ps)\ rs)$

lemma *remdups-rev-code*[code-unfold]: $remdups\text{-}rev\ rs = remdups\text{-}rev\text{-}code\ []\ rs$
 <proof>

end

theory *Ipassmt*

imports *Common-Primitive-Syntax*

../Semantics-Ternary/Primitive-Normalization

Simple-Firewall.Iface

Simple-Firewall.IP-Addr-WordInterval-toString

Automatic-Refinement.Misc

begin

hide-const *Misc.uncurry*

hide-fact *Misc.uncurry-def*

A mapping from an interface to its assigned ip addresses in CIDR notation

type-synonym *'i ipassignment=iface* \rightarrow (*'i word* \times *nat*) *list*

31.1 Sanity checking for an *'i ipassignment*.

warning if interface map has wildcards

definition *ipassmt-sanity-nowildcards* :: *'i ipassignment* \Rightarrow **bool** **where**
ipassmt-sanity-nowildcards ipassmt $\equiv \forall\ iface \in dom\ ipassmt.\ \neg\ iface\text{-}is\text{-}wildcard\ iface$

Executable of the *'i ipassignment* is given as a list.

lemma[code-unfold]: *ipassmt-sanity-nowildcards (map-of ipassmt)* $\longleftrightarrow (\forall\ iface \in fst'\ set\ ipassmt.\ \neg\ iface\text{-}is\text{-}wildcard\ iface)$
 <proof>

lemma *ipassmt-sanity-nowildcards-match-iface*:

ipassmt-sanity-nowildcards ipassmt \Longrightarrow

ipassmt (Iface ifce2) = None \Longrightarrow

ipassmt ifce = Some a \Longrightarrow

$\neg\ match\text{-}iface\ ifce\ ifce2$

<proof>

definition *map-of-ipassmt* :: (*iface* \times (*'i word* \times *nat*) *list*) *list* \Rightarrow *iface* \rightarrow (*'i word* \times *nat*) *list* **where**

map-of-ipassmt ipassmt = (

if

distinct (map fst ipassmt) \wedge ipassmt-sanity-nowildcards (map-of ipassmt)

then
 map-of ipassmt
 else undefined ~~undefined ipassmt must be distinct and don't have wildcard interfaces~~

some additional (optional) sanity checks

sanity check that there are no zone-spanning interfaces

definition *ipassmt-sanity-disjoint* :: 'i::len ipassignment ⇒ bool **where**
ipassmt-sanity-disjoint ipassmt ≡ ∀ i1 ∈ dom ipassmt. ∀ i2 ∈ dom ipassmt. i1
 ≠ i2 →
ipcidr-union-set (set (the (ipassmt i1))) ∩ *ipcidr-union-set (set (the (ipassmt
 i2)))* = {}

lemma[code-unfold]: *ipassmt-sanity-disjoint (map-of ipassmt)* ↔
 (let Is = fst' set ipassmt in
 (∀ i1 ∈ Is. ∀ i2 ∈ Is. i1 ≠ i2 → wordinterval-empty (wordinterval-intersection
 (l2wi (map ipcidr-to-interval (the ((map-of ipassmt) i1)))) (l2wi (map ipcidr-to-interval
 (the ((map-of ipassmt) i2)))))))
 ⟨proof⟩

Checking that the ipassmt covers the complete ipv4 address space.

definition *ipassmt-sanity-complete* :: (iface × ('i::len word × nat) list) list ⇒
 bool **where**
ipassmt-sanity-complete ipassmt ≡ distinct (map fst ipassmt) ∧ (∪ (ipcidr-union-set
 'set ' (ran (map-of ipassmt)))) = UNIV

lemma[code-unfold]: *ipassmt-sanity-complete ipassmt* ↔ distinct (map fst
 ipassmt) ∧ (let range = map snd ipassmt in
 wordinterval-eq (wordinterval-Union (map (l2wi ∘ (map ipcidr-to-interval))
 range)) wordinterval-UNIV
)
 ⟨proof⟩

value[code] *ipassmt-sanity-nowildcards* (map-of [(Iface "eth1.1017", [(ipv4addr-of-dotdecimal
 (131,159,14,240), 28)]))])

fun *collect-ifaces'* :: 'i::len common-primitive rule list ⇒ iface list **where**
collect-ifaces' [] = [] |
collect-ifaces' ((Rule m a)#rs) = filter (λiface. iface ≠ ifaceAny) (
 (map (λx. case x of Pos i ⇒ i | Neg i ⇒ i) (fst
 (primitive-extractor (is-Iiface, iface-sel) m))) @
 (map (λx. case x of Pos i ⇒ i | Neg i ⇒ i) (fst
 (primitive-extractor (is-Oiface, oiface-sel) m))) @ *collect-ifaces'* rs)

definition *collect-ifaces* :: 'i::len common-primitive rule list ⇒ iface list **where**
collect-ifaces rs ≡ mergesort-remdups (*collect-ifaces'* rs)

lemma set (*collect-ifaces* rs) = set (*collect-ifaces'* rs)

$\langle \text{proof} \rangle$

sanity check that all interfaces mentioned in the ruleset are also listed in the ipassmt. May fail for wildcard interfaces in the ruleset.

definition *ipassmt-sanity-defined* :: 'i::len common-primitive rule list \Rightarrow 'i ipassignment \Rightarrow bool **where**

ipassmt-sanity-defined rs ipassmt $\equiv \forall$ iface \in set (collect-ifaces rs). iface \in dom ipassmt

lemma[code]: *ipassmt-sanity-defined* rs ipassmt $\longleftrightarrow (\forall$ iface \in set (collect-ifaces rs). ipassmt iface \neq None)

$\langle \text{proof} \rangle$

lemma *ipassmt-sanity-defined* [

Rule (MatchAnd (Match (Src (IpAddrNetmask (ipv4addr-of-dotdecimal (192,168,0,0) 24))) (Match (Iiface (Iface "eth1.1017")))) action.Accept,
Rule (MatchAnd (Match (Src (IpAddrNetmask (ipv4addr-of-dotdecimal (192,168,0,0) 24))) (Match (Iiface (ifaceAny)))) action.Accept,
Rule MatchAny action.Drop]
(map-of [(Iface "eth1.1017", [(ipv4addr-of-dotdecimal (131,159,14,240), 28)]))]) $\langle \text{proof} \rangle$

definition *ipassmt-ignore-wildcard* :: 'i::len ipassignment \Rightarrow 'i ipassignment **where**
ipassmt-ignore-wildcard ipassmt $\equiv \lambda k$. case ipassmt k of None \Rightarrow None

| Some ips \Rightarrow if ipcidr-union-set

(set ips) = UNIV then None else Some ips

lemma *ipassmt-ignore-wildcard-le*: *ipassmt-ignore-wildcard* ipassmt \subseteq_m ipassmt
 $\langle \text{proof} \rangle$

definition *ipassmt-ignore-wildcard-list*:: (iface \times ('i::len word \times nat) list) list \Rightarrow (iface \times ('i word \times nat) list) list **where**

ipassmt-ignore-wildcard-list ipassmt = filter ($\lambda(-, ips)$. \neg wordinterval-eq (l2wi (map ipcidr-to-interval ips)) wordinterval-UNIV) ipassmt

lemma *distinct* (map fst ipassmt) \Longrightarrow

map-of (*ipassmt-ignore-wildcard-list* ipassmt) = *ipassmt-ignore-wildcard* (map-of ipassmt)

$\langle \text{proof} \rangle$

Debug algorithm with human-readable output

definition *debug-ipassmt-generic*

:: ('i::len wordinterval \Rightarrow string) \Rightarrow

(iface \times ('i word \times nat) list) list \Rightarrow 'i common-primitive rule list \Rightarrow string list **where**

```

debug-ipassmt-generic toStr ipassmt rs ≡ let ifaces = (map fst ipassmt) in [
  "distinct: " @ (if distinct ifaces then "passed" else "FAIL!")
  , "ipassmt-sanity-nowildcards: " @
    (if ipassmt-sanity-nowildcards (map-of ipassmt)
     then "passed" else "fail: "@list-toString iface-sel (filter iface-is-wildcard
ifaces))
  , "ipassmt-sanity-defined (interfaces defined in the ruleset are also in ipassmt):
" @
    (if ipassmt-sanity-defined rs (map-of ipassmt)
     then "passed" else "fail: "@list-toString iface-sel [i ← (collect-ifaces rs).
i ∉ set ifaces])
  , "ipassmt-sanity-disjoint (no zone-spanning interfaces): " @
    (if ipassmt-sanity-disjoint (map-of ipassmt)
     then "passed" else "fail: "@list-toString (λ(i1,i2). "(" @ iface-sel i1 @
", " @ iface-sel i2 @ ")")
      [(i1,i2) ← List.product ifaces ifaces. i1 ≠ i2 ∧
       ¬ wordinterval-empty (wordinterval-intersection
                             (l2wi (map ipcidr-to-interval (the ((map-of ipassmt)
i1))))
                             (l2wi (map ipcidr-to-interval (the ((map-of ipassmt)
i2))))))
    ]
  , "ipassmt-sanity-disjoint excluding UNIV interfaces: " @
    (let ipassmt = ipassmt-ignore-wildcard-list ipassmt;
      ifaces = (map fst ipassmt)
      in
      (if ipassmt-sanity-disjoint (map-of ipassmt)
       then "passed" else "fail: "@list-toString (λ(i1,i2). "(" @ iface-sel i1 @
", " @ iface-sel i2 @ ")")
        [(i1,i2) ← List.product ifaces ifaces. i1 ≠ i2 ∧
         ¬ wordinterval-empty (wordinterval-intersection
                               (l2wi (map ipcidr-to-interval (the ((map-of ipassmt)
i1))))
                               (l2wi (map ipcidr-to-interval (the ((map-of ipassmt)
i2))))))
        ]))
  , "ipassmt-sanity-complete: " @
    (if ipassmt-sanity-complete ipassmt
     then "passed"
     else "the following is not covered: " @
      toStr (wordinterval-setminus wordinterval-UNIV (wordinterval-Union
(map (l2wi ∘ (map ipcidr-to-interval)) (map snd ipassmt))))))
  , "ipassmt-sanity-complete excluding UNIV interfaces: " @
    (let ipassmt = ipassmt-ignore-wildcard-list ipassmt
      in
      (if ipassmt-sanity-complete ipassmt
       then "passed"
       else "the following is not covered: " @
        toStr (wordinterval-setminus wordinterval-UNIV (wordinterval-Union

```

(map (l2wi ◦ (map ipcidr-to-interval)) (map snd ipassmt))))))
]

definition *debug-ipassmt-ipv4* ≡ *debug-ipassmt-generic ipv4addr-wordinterval-toString*

definition *debug-ipassmt-ipv6* ≡ *debug-ipassmt-generic ipv6addr-wordinterval-toString*

lemma *dom-ipassmt-ignore-wildcard*:

$i \in \text{dom } (\text{ipassmt-ignore-wildcard } \text{ipassmt}) \iff i \in \text{dom } \text{ipassmt} \wedge \text{ipcidr-union-set } (\text{set } (\text{the } (\text{ipassmt } i))) \neq \text{UNIV}$
 <proof>

lemma *ipassmt-ignore-wildcard-the*:

$\text{ipassmt } i = \text{Some } \text{ips} \implies \text{ipcidr-union-set } (\text{set } \text{ips}) \neq \text{UNIV} \implies (\text{the } (\text{ipassmt-ignore-wildcard } \text{ipassmt } i)) = \text{ips}$
 $\text{ipassmt-ignore-wildcard } \text{ipassmt } i = \text{Some } \text{ips} \implies \text{the } (\text{ipassmt } i) = \text{ips}$
 $\text{ipassmt-ignore-wildcard } \text{ipassmt } i = \text{Some } \text{ips} \implies \text{ipcidr-union-set } (\text{set } \text{ips}) \neq \text{UNIV}$
 <proof>

lemma *ipassmt-sanity-disjoint-ignore-wildcards*:

$\text{ipassmt-sanity-disjoint } (\text{ipassmt-ignore-wildcard } \text{ipassmt}) \iff$
 $(\forall i1 \in \text{dom } \text{ipassmt}.$
 $\forall i2 \in \text{dom } \text{ipassmt}.$
 $\text{ipcidr-union-set } (\text{set } (\text{the } (\text{ipassmt } i1))) \neq \text{UNIV} \wedge$
 $\text{ipcidr-union-set } (\text{set } (\text{the } (\text{ipassmt } i2))) \neq \text{UNIV} \wedge$
 $i1 \neq i2$
 $\implies \text{ipcidr-union-set } (\text{set } (\text{the } (\text{ipassmt } i1))) \cap \text{ipcidr-union-set } (\text{set } (\text{the } (\text{ipassmt } i2))) = \{\}$)
 <proof>

Confusing names: *ipassmt-sanity-nowildcards* refers to wildcard interfaces.
ipassmt-ignore-wildcard refers to the UNIV ip range.

lemma *ipassmt-sanity-nowildcards-ignore-wildcardD*:

$\text{ipassmt-sanity-nowildcards } \text{ipassmt} \implies \text{ipassmt-sanity-nowildcards } (\text{ipassmt-ignore-wildcard } \text{ipassmt})$
 <proof>

lemma *ipassmt-disjoint-nonempty-inj*:

assumes *ipassmt-disjoint*: *ipassmt-sanity-disjoint ipassmt*
and *ifce*: *ipassmt ifce = Some i-ips*
and *a*: *ipcidr-union-set (set i-ips) ≠ {}*
and *k*: *ipassmt k = Some i-ips*
shows *k = ifce*
 <proof>

lemma *ipassmt-ignore-wildcard-None-Some*:

ipassmt-ignore-wildcard ipassmt ifce = None \implies *ipassmt ifce = Some ips* \implies
ipcidr-union-set (set ips) = UNIV
 ⟨proof⟩

lemma *ipassmt-disjoint-ignore-wildcard-nonempty-inj*:
assumes *ipassmt-disjoint: ipassmt-sanity-disjoint (ipassmt-ignore-wildcard ipassmt)*
and *ifce: ipassmt ifce = Some i-ips*
and *a: ipcidr-union-set (set i-ips) \neq {}*
and *k: (ipassmt-ignore-wildcard ipassmt) k = Some i-ips*
shows *k = ifce*
 ⟨proof⟩

lemma *ipassmt-disjoint-inj-k*:
assumes *ipassmt-disjoint: ipassmt-sanity-disjoint ipassmt*
and *ifce: ipassmt ifce = Some ips*
and *k: ipassmt k = Some ips'*
and *a: p \in ipcidr-union-set (set ips)*
and *b: p \in ipcidr-union-set (set ips')*
shows *k = ifce*
 ⟨proof⟩

lemma *ipassmt-disjoint-matcheq-iiface-srcip*:
assumes *ipassmt-nowild: ipassmt-sanity-nowildcards ipassmt*
and *ipassmt-disjoint: ipassmt-sanity-disjoint ipassmt*
and *ifce: ipassmt ifce = Some i-ips*
and *p-ifce: ipassmt (Iface (p-iiface p)) = Some p-ips \wedge p-src p \in ipcidr-union-set (set p-ips)*
shows *match-iface ifce (p-iiface p) \longleftrightarrow p-src p \in ipcidr-union-set (set i-ips)*
 ⟨proof⟩

definition *ipassmt-generic-ipv4* :: (*iface* \times (32 word \times nat) list) list **where**
ipassmt-generic-ipv4 = [(Iface "lo", [(ipv4addr-of-dotdecimal (127,0,0,0),8)])]

definition *ipassmt-generic-ipv6* :: (*iface* \times (128 word \times nat) list) list **where**
ipassmt-generic-ipv6 = [(Iface "lo", [(1,128)])]

31.2 IP Assignment difference

Compare two ipassmts. Returns a list of tuples. First entry of the tuple: things which are in the left ipassmt but not in the right. Second entry of the tuple: things which are in the right ipassmt but not in the left.

definition *ipassmt-diff*

```

:: (iface × ('i::len word × nat) list) list ⇒ (iface × ('i::len word × nat) list)
list
  ⇒ (iface × ('i word × nat) list × ('i word × nat) list) list
where
  ipassmt-diff a b ≡ let
    t = λs. (case s of None ⇒ Empty-WordInterval
              | Some s ⇒ wordinterval-Union (map ipcidr-tuple-to-wordinterval
s));
    k = λx y d. cidr-split (wordinterval-setminus (t (map-of x d)) (t (map-of y
d)))
  in
    [(d, (k a b d, k b a d)). d ← remdups (map fst (a @ b))]

```

If an interface is defined in both ipassignments and there is no difference then the two ipassignments describe the same IP range for this interface.

lemma *ipassmt-diff-ifce-equal*: $(ifce, [], []) \in \text{set } (\text{ipassmt-diff } ipassmt1 \text{ } ipassmt2)$
 \implies
 $ifce \in \text{dom } (\text{map-of } ipassmt1) \implies ifce \in \text{dom } (\text{map-of } ipassmt2) \implies$
 $ipcidr\text{-union-set } (\text{set } (\text{the } ((\text{map-of } ipassmt1) \text{ } ifce))) =$
 $ipcidr\text{-union-set } (\text{set } (\text{the } ((\text{map-of } ipassmt2) \text{ } ifce)))$
<proof>

lemma *ipcidr-union-cidr-split[simp]*: $ipcidr\text{-union-set } (\text{set } (\text{cidr-split } a)) = \text{wordinter-}$
 $\text{val-to-set } a$
<proof>

lemma

defines *assmt as ifce* ≡ $ipcidr\text{-union-set } (\text{set } (\text{the } ((\text{map-of } as \text{ } ifce))))$
assumes *diffs*: $(ifce, d1, d2) \in \text{set } (\text{ipassmt-diff } ipassmt1 \text{ } ipassmt2)$
and *doms*: $ifce \in \text{dom } (\text{map-of } ipassmt1) \text{ } ifce \in \text{dom } (\text{map-of } ipassmt2)$
shows $ipcidr\text{-union-set } (\text{set } d1) = \text{assmt } ipassmt1 \text{ } ifce - \text{assmt } ipassmt2 \text{ } ifce$
 $ipcidr\text{-union-set } (\text{set } d2) = \text{assmt } ipassmt2 \text{ } ifce - \text{assmt } ipassmt1 \text{ } ifce$
<proof>

Explanation for interface *Iface "a"*: Left ipassmt: The IP range 4/30 contains the addresses 4,5,6,7 Diff: right ipassmt contains 6/32, so 4,5,7 is only in the left ipassmt. IP addresses 4,5 correspond to subnet 4/30.

lemma *ipassmt-diff (ipassmt-generic-ipv4 @ [(Iface "a", [(4,30)]))*
 $(ipassmt\text{-generic-ipv4 } @ [(Iface \text{ "a"}, [(6,32), (0,30)]), (Iface$
 $\text{ "b"}, [(42,32)])) =$
 $[(Iface \text{ "lo"}, [], []),$
 $(Iface \text{ "a"}, [(4, 31),(7, 32)],$
 $[(0, 30)$
 $),$
 $(Iface \text{ "b"}, [], [(42, 32))]]$ *<proof>*

end

theory *No-Spoof*

imports *Common-Primitive-Lemmas*

Ipassmt
begin

32 No Spoofing

assumes: *simple-ruleset*

32.1 Spoofing Protection

No spoofing means: Every packet that is (potentially) allowed by the firewall and comes from an interface *iface* must have a Source IP Address in the assigned range *iface*.

“potentially allowed” means we use the upper closure. The definition states: For all interfaces which are configured, every packet that comes from this interface and is allowed by the firewall must be in the IP range of that interface.

We add *'pkt-ext itself* as a parameter to have the type of a generic, extensible packet in the definition.

definition *no-spoofing* :: *'pkt-ext itself* \Rightarrow *'i::len ipassignment* \Rightarrow *'i::len common-primitive rule list* \Rightarrow *bool* **where**
no-spoofing TYPE(*'pkt-ext*) *ipassmt* *rs* \equiv \forall *iface* \in *dom ipassmt*. \forall *p* :: (*'i, 'pkt-ext*) *tagged-packet-scheme*.
 $((\text{common-matcher}, \text{in-doubt-allow}), p(p\text{-iface}:=\text{iface-sel } \text{iface})) \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow} \longrightarrow$
 $p\text{-src } p \in (\text{ipcidr-union-set } (\text{set } (\text{the } (\text{ipassmt } \text{iface}))))$

This is how it looks like for an IPv4 simple packet: We add *unit* because a *32 tagged-packet* does not have any additional fields.

lemma *no-spoofing* TYPE(*unit*) *ipassmt* *rs* \longleftrightarrow
 $(\forall$ *iface* \in *dom ipassmt*. \forall *p* :: *32 tagged-packet*.
 $((\text{common-matcher}, \text{in-doubt-allow}), p(p\text{-iface}:=\text{iface-sel } \text{iface})) \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow} \longrightarrow$
 $p\text{-src } p \in (\text{ipcidr-union-set } (\text{set } (\text{the } (\text{ipassmt } \text{iface}))))$
 $\langle \text{proof} \rangle$

The definition is sound (if that can be said about a definition): if *no-spoofing* certifies your ruleset, then your ruleset prohibits spoofing. The definition may not be complete: *no-spoofing* may return *False* even though your ruleset prevents spoofing (should only occur if some strange and unknown primitives occur)

Technical note: The definition can be thought of as protection from OUTGOING spoofing. OUTGOING means: I define my interfaces and their IP addresses. For all interfaces, only the assigned IP addresses may pass the

firewall. This definition is simple for e.g. local sub-networks. Example: $[Iface\ "eth0" \mapsto \{(ip\ v4\ addr\ of\ dot\ decimal\ (192,\ 168,\ 0,\ 0),\ 24::'a)\}]$

If I want spoofing protection from the Internet, I need to specify the range of the Internet IP addresses. Example: $[Iface\ "evil-internet" \mapsto \{everything\ that\ does\ not\ belong\ to\ me\}]$

This is also a good opportunity to exclude the private IP space, link local, and probably multicast space. See *all-but-those-ips* to easily specify these ranges.

See examples below. Check Example 3 why it can be thought of as OUT-GOING spoofing.

context
begin

The set of any ip addresses which may match for a fixed *iface* (overapproximation)

private definition *get-exists-matching-src-ips* :: *iface* \Rightarrow *'i::len common-primitive match-expr* \Rightarrow *'i word set* **where**
get-exists-matching-src-ips *iface m* \equiv *let* (*i-matches*, *-*) = (*primitive-extractor* (*is-Iiface*, *iiface-sel*) *m*) *in*
if (\forall *is* \in *set i-matches*. (*case is of Pos i* \Rightarrow *match-iface i* (*iface-sel* *iface*)
| *Neg i* \Rightarrow \neg *match-iface i* (*iface-sel* *iface*)))
then
(let (*ip-matches*, *-*) = (*primitive-extractor* (*is-Src*, *src-sel*) *m*) *in*
if *ip-matches* = []
then
UNIV
else
 \bigcap *ips* \in *set (ip-matches)*. (*case ips of Pos ip* \Rightarrow *ipt-irange-to-set* *ip* | *Neg ip* \Rightarrow \neg *ipt-irange-to-set ip*)
else
{}
end

lemma *primitive-extractor* (*is-Src*, *src-sel*)
(*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (0::*ip\ v4\ addr*) 30))) (*Match* (*Iiface* (*Iface* "eth0")))) =
(*[Pos* (*IpAddrNetmask* 0 30)], *MatchAnd MatchAny* (*Match* (*Iiface* (*Iface* "eth0")))) *<proof>* **lemma** *get-exists-matching-src-ips-subset*:
assumes *normalized-nnf-match m*
shows $\{ip. (\exists p :: ('i::len, 'a) \text{ tagged-packet-scheme. matches (common-matcher, in-doubt-allow) } m \ a \ (p\ \backslash\ p\text{-iiface} := \text{iface-sel } \text{iface}, \ p\text{-src} := ip))\} \subseteq$
get-exists-matching-src-ips *iface m*
<proof>

lemma *common-primitive-not-has-primitive-expand*:
 \neg *has-primitive* (*m*::*'i::len common-primitive match-expr*) \longleftrightarrow

\neg *has-disc is-Dst* *m* \wedge
 \neg *has-disc is-Src* *m* \wedge
 \neg *has-disc is-Iiface* *m* \wedge
 \neg *has-disc is-Oiface* *m* \wedge
 \neg *has-disc is-Prot* *m* \wedge
 \neg *has-disc is-Src-Ports* *m* \wedge
 \neg *has-disc is-Dst-Ports* *m* \wedge
 \neg *has-disc is-MultiportPorts* *m* \wedge
 \neg *has-disc is-L4-Flags* *m* \wedge
 \neg *has-disc is-CT-State* *m* \wedge
 \neg *has-disc is-Extra* *m*
 ⟨*proof*⟩

lemma \neg *has-primitive* *m* \wedge *matcheq-matchAny* *m* \longleftrightarrow (if \neg *has-primitive* *m*
 then *matcheq-matchAny* *m* else *False*)
 ⟨*proof*⟩

The set of ip addresses which definitely match for a fixed *iface* (underapproximation)

private definition *get-all-matching-src-ips* :: *iface* \Rightarrow '*i*::len *common-primitive*
match-expr \Rightarrow '*i* word set **where**
get-all-matching-src-ips *iface* *m* \equiv let (*i-matches*, *rest1*) = (*primitive-extractor*
 (*is-Iiface*, *iface-sel*) *m*) in
 if (\forall *is* \in set *i-matches*. (case *is* of *Pos* *i* \Rightarrow *match-iface* *i* (*iface-sel*
 iface)
 | *Neg* *i* \Rightarrow \neg *match-iface* *i* (*iface-sel* *iface*)))
 then
 (let (*ip-matches*, *rest2*) = (*primitive-extractor* (*is-Src*, *src-sel*) *rest1*)
 in
 if \neg *has-primitive* *rest2* \wedge *matcheq-matchAny* *rest2*
 then
 if *ip-matches* = []
 then
 UNIV
 else
 \bigcap *ips* \in set (*ip-matches*). (case *ips* of *Pos* *ip* \Rightarrow *ipt-iprange-to-set*
 ip | *Neg* *ip* \Rightarrow \neg *ipt-iprange-to-set* *ip*)
 else
 { })
 else
 { }

private lemma *get-all-matching-src-ips*:
 assumes *normalized-nnf-match* *m*
 shows *get-all-matching-src-ips* *iface* *m* \subseteq

```

      {ip. (∀ p::('i::len, 'a) tagged-packet-scheme. matches (common-matcher,
in-doubt-allow) m a (p(p-iiface:= iface-sel iface, p-src:= ip)))}
    ⟨proof⟩ definition get-exists-matching-src-ips-executable
      :: iface ⇒ 'i::len common-primitive match-expr ⇒ 'i wordinterval where
        get-exists-matching-src-ips-executable iface m ≡ let (i-matches, -) = (primitive-extractor
(is-Iiface, iiface-sel) m) in
          if (∀ is ∈ set i-matches. (case is of Pos i ⇒ match-iface i (iface-sel
iface)
                                     | Neg i ⇒ ¬match-iface i (iface-sel iface)))
            then
              (let (ip-matches, -) = (primitive-extractor (is-Src, src-sel) m) in
               if ip-matches = []
                 then
                   wordinterval-UNIV
                 else
                   l2wi-negation-type-intersect (NegPos-map ipt-iprange-to-interval
ip-matches))
            else
              Empty-WordInterval

```

lemma *get-exists-matching-src-ips-executable*:

```

wordinterval-to-set (get-exists-matching-src-ips-executable iface m) = get-exists-matching-src-ips
iface m
⟨proof⟩

```

lemma (*get-exists-matching-src-ips-executable* (Iface "eth0"))

```

(MatchAnd (MatchNot (Match (Src (IpAddrNetmask (ipv4addr-of-dotdecimal
(192,168,0,0) 24)))) (Match (Iiface (Iface "eth0"))))) =

```

```

RangeUnion (WordInterval 0 0xC0A7FFFF) (WordInterval 0xC0A80100
0xFFFFFFFF) ⟨proof⟩ definition get-all-matching-src-ips-executable

```

```

:: iface ⇒ 'i::len common-primitive match-expr ⇒ 'i wordinterval where
  get-all-matching-src-ips-executable iface m ≡ let (i-matches, rest1) = (primitive-extractor
(is-Iiface, iiface-sel) m) in

```

```

    if (∀ is ∈ set i-matches. (case is of Pos i ⇒ match-iface i (iface-sel
iface)
                               | Neg i ⇒ ¬match-iface i (iface-sel iface)))
      then

```

```

        (let (ip-matches, rest2) = (primitive-extractor (is-Src, src-sel) rest1)

```

in

```

        if ¬ has-primitive rest2 ∧ matcheq-matchAny rest2

```

```

        then
          if ip-matches = []

```

```

            then
              wordinterval-UNIV
            else

```

```

              l2wi-negation-type-intersect (NegPos-map ipt-iprange-to-interval
ip-matches)

```

```

            else
              Empty-WordInterval)

```

else
 Empty-WordInterval

lemma *get-all-matching-src-ips-executable*:
 wordinterval-to-set (get-all-matching-src-ips-executable iface m) = get-all-matching-src-ips
 iface m
 ⟨proof⟩

lemma (get-all-matching-src-ips-executable (Iface "eth0"))
 (MatchAnd (MatchNot (Match (Src (IpAddrNetmask (ipv4addr-of-dotdecimal
 (192,168,0,0) 24)))) (Match (Iiface (Iface "eth0"))))) =
 RangeUnion (WordInterval 0 0xC0A7FFFF) (WordInterval 0xC0A80100
 0xFFFFFFFF) ⟨proof⟩

The following algorithm sound but not complete.

private fun *no-spoofing-algorithm*
 :: iface ⇒ 'i::len ipassignment ⇒ 'i common-primitive rule list ⇒ 'i word set
 ⇒ 'i word set ⇒ bool **where**
 no-spoofing-algorithm iface ipassmt [] allowed denied1 ←→
 (allowed – denied1) ⊆ ipcidr-union-set (set (the (ipassmt iface))) |
 no-spoofing-algorithm iface ipassmt ((Rule m Accept)#rs) allowed denied1 =
 no-spoofing-algorithm iface ipassmt rs
 (allowed ∪ get-exists-matching-src-ips iface m) denied1 |
 no-spoofing-algorithm iface ipassmt ((Rule m Drop)#rs) allowed denied1 =
 no-spoofing-algorithm iface ipassmt rs
 allowed (denied1 ∪ (get-all-matching-src-ips iface m – allowed)) |
 no-spoofing-algorithm - - - - = undefined

private fun *no-spoofing-algorithm-executable*
 :: iface ⇒ (iface → ('i::len word × nat) list) ⇒ 'i common-primitive rule list
 ⇒ 'i wordinterval ⇒ 'i wordinterval ⇒ bool **where**
 no-spoofing-algorithm-executable iface ipassmt [] allowed denied1 ←→
 wordinterval-subset (wordinterval-setminus allowed denied1) (l2wi (map ip-
 cidr-to-interval (the (ipassmt iface)))) |
 no-spoofing-algorithm-executable iface ipassmt ((Rule m Accept)#rs) allowed
 denied1 = no-spoofing-algorithm-executable iface ipassmt rs
 (wordinterval-union allowed (get-exists-matching-src-ips-executable iface m))
 denied1 |
 no-spoofing-algorithm-executable iface ipassmt ((Rule m Drop)#rs) allowed de-
 nied1 = no-spoofing-algorithm-executable iface ipassmt rs
 allowed (wordinterval-union denied1 (wordinterval-setminus (get-all-matching-src-ips-executable
 iface m) allowed)) |
 no-spoofing-algorithm-executable - - - - = undefined

lemma *no-spoofing-algorithm-executable*: no-spoofing-algorithm-executable iface
 ipassmt rs allowed denied ←→
 no-spoofing-algorithm iface ipassmt rs (wordinterval-to-set allowed) (wordinterval-to-set
 denied)

<proof> **definition** *nospoof* $\text{TYPE}('pkt\text{-}ext)$ *iface* *ipassmt* *rs* = $(\forall p :: ('i::len, 'pkt\text{-}ext)$
tagged-packet-scheme.
 $(\text{approximating-bigstep-fun } (common\text{-}matcher, in\text{-}doubt\text{-}allow) (p(p\text{-}iface:=iface\text{-}sel$
iface))) *rs* *Undecided* = *Decision* *FinalAllow*) \longrightarrow
 $p\text{-}src$ $p \in (ipcidr\text{-}union\text{-}set (set (the (ipassmt\text{-}iface))))$)
private definition *setbydecision* $\text{TYPE}('pkt\text{-}ext)$ *iface* *rs* *dec* = $\{ip. \exists p :: ('i::len, 'pkt\text{-}ext)$
tagged-packet-scheme. approximating-bigstep-fun (common-matcher, in-doubt-allow)

$(p(p\text{-}iface:=iface\text{-}sel\text{-}iface, p\text{-}src := ip))$ *rs* *Undecided* =
Decision *dec*}

private lemma *nospoof-setbydecision*:

fixes *rs* :: *'i::len* *common-primitive* *rule* *list*

shows *nospoof* $\text{TYPE}('pkt\text{-}ext)$ *iface* *ipassmt* *rs* \longleftrightarrow

$setbydecision\text{-}all\text{-}TYPE('pkt\text{-}ext)$ *iface* *rs* *FinalAllow* $\subseteq (ipcidr\text{-}union\text{-}set (set$
 $(the (ipassmt\text{-}iface))))$)

<proof> **definition** *setbydecision-all* $\text{TYPE}('pkt\text{-}ext)$ *iface* *rs* *dec* = $\{ip. \forall p ::$
 $('i::len, 'pkt\text{-}ext)$ *tagged-packet-scheme.*

$approximating-bigstep-fun (common\text{-}matcher, in\text{-}doubt\text{-}allow) (p(p\text{-}iface:=iface\text{-}sel$
*iface, p-src := ip)) *rs* *Undecided* = *Decision* *dec*}*

private lemma *setbydecision-setbydecision-all-Allow*:

$(setbydecision\text{-}TYPE('pkt\text{-}ext)$ *iface* *rs* *FinalAllow* – *setbydecision-all* $\text{TYPE}('pkt\text{-}ext)$
iface *rs* *FinalDeny*) =

$setbydecision\text{-}TYPE('pkt\text{-}ext)$ *iface* *rs* *FinalAllow*

<proof> **lemma** *setbydecision-setbydecision-all-Deny*:

$(setbydecision\text{-}TYPE('pkt\text{-}ext)$ *iface* *rs* *FinalDeny* – *setbydecision-all* $\text{TYPE}('pkt\text{-}ext)$
iface *rs* *FinalAllow*) =

$setbydecision\text{-}TYPE('pkt\text{-}ext)$ *iface* *rs* *FinalDeny*

<proof> **lemma** *setbydecision-append*:

simple-ruleset (*rs1* @ *rs2*) \implies

$setbydecision\text{-}TYPE('pkt\text{-}ext)$ *iface* (*rs1* @ *rs2*) *FinalAllow* =

$setbydecision\text{-}TYPE('pkt\text{-}ext)$ *iface* *rs1* *FinalAllow* $\cup \{ip. \exists p :: ('i::len, 'pkt\text{-}ext)$
tagged-packet-scheme. approximating-bigstep-fun (common-matcher, in-doubt-allow)

$(p(p\text{-}iface:=iface\text{-}sel\text{-}iface, p\text{-}src := ip))$ *rs2* *Undecided* = *Decision* *FinalAl-*
low \wedge

$approximating-bigstep-fun (common\text{-}matcher, in\text{-}doubt\text{-}allow) (p(p\text{-}iface:=iface\text{-}sel$
*iface, p-src := ip)) *rs1* *Undecided* = *Undecided*}*

<proof> **lemma** *not-FinalAllow*: *foo* \neq *Decision* *FinalAllow* \longleftrightarrow *foo* = *Decision*
FinalDeny \vee *foo* = *Undecided*

<proof> **lemma** *setbydecision-all-appendAccept*: *simple-ruleset* (*rs* @ [*Rule* *r*
Accept]) \implies

$setbydecision\text{-}all\text{-}TYPE('pkt\text{-}ext)$ *iface* *rs* *FinalDeny* = $setbydecision\text{-}all\text{-}TYPE('pkt\text{-}ext)$
iface (*rs* @ [*Rule* *r* *Accept*]) *FinalDeny*

<proof> **lemma** *setbydecision-all-append-subset*: *simple-ruleset* (*rs1* @ *rs2*)

\implies

$setbydecision\text{-}all\text{-}TYPE('pkt\text{-}ext)$ *iface* *rs1* *FinalDeny* $\cup \{ip. \forall p ::$
 $('i::len, 'pkt\text{-}ext)$ *tagged-packet-scheme.*

$\text{approximating-bigstep-fun } (\text{common-matcher}, \text{in-doubt-allow}) (p \setminus p\text{-iface} := \text{iface-sel } \text{iface}, p\text{-src} := ip) \setminus rs2 \text{ Undecided} = \text{Decision FinalDeny} \wedge$
 $\text{approximating-bigstep-fun } (\text{common-matcher}, \text{in-doubt-allow}) (p \setminus p\text{-iface} := \text{iface-sel } \text{iface}, p\text{-src} := ip) \setminus rs1 \text{ Undecided} = \text{Undecided}$

\subseteq
 $\text{setbydecision-all TYPE('pkt-ext) iface } (rs1 @ rs2) \text{ FinalDeny}$
 $\langle \text{proof} \rangle \text{ lemma setbydecision-all TYPE('pkt-ext) iface } rs1 \text{ FinalDeny} \cup$
 $\{ip. \forall p :: ('i::\text{len}, 'pkt\text{-ext}) \text{ tagged-packet-scheme.}$
 $\text{approximating-bigstep-fun } (\text{common-matcher}, \text{in-doubt-allow}) (p \setminus p\text{-iface} := \text{iface-sel } \text{iface}, p\text{-src} := ip) \setminus rs1 \text{ Undecided} = \text{Undecided}$

\subseteq
 $-\text{ setbydecision TYPE('pkt-ext) iface } rs1 \text{ FinalAllow}$
 $\langle \text{proof} \rangle \text{ lemma Collect-minus-eq: } \{x. P x\} - \{x. Q x\} = \{x. P x \wedge \neg Q x\}$
 $\langle \text{proof} \rangle \text{ lemma setbydecision-all-append-subset2:}$
 $\text{simple-ruleset } (rs1 @ rs2) \implies$
 $\text{setbydecision-all TYPE('pkt-ext) iface } rs1 \text{ FinalDeny} \cup$
 $(\text{setbydecision-all TYPE('pkt-ext) iface } rs2 \text{ FinalDeny} -$
 $\text{setbydecision TYPE('pkt-ext) iface } rs1 \text{ FinalAllow})$
 $\subseteq \text{setbydecision-all TYPE('pkt-ext) iface } (rs1 @ rs2) \text{ FinalDeny}$
 $\langle \text{proof} \rangle \text{ lemma setbydecision-all TYPE('pkt-ext) iface } rs \text{ FinalDeny} \subseteq -$
 $\text{setbydecision TYPE('pkt-ext) iface } rs \text{ FinalAllow}$
 $\langle \text{proof} \rangle \text{ lemma no-spoofing-algorithm-sound-generalized:}$
fixes $rs1 :: 'i::\text{len} \text{ common-primitive rule list}$
shows $\text{simple-ruleset } rs1 \implies \text{simple-ruleset } rs2 \implies$
 $(\forall r \in \text{set } rs2. \text{normalized-nnf-match } (\text{get-match } r)) \implies$
 $\text{setbydecision TYPE('pkt-ext) iface } rs1 \text{ FinalAllow} \subseteq \text{allowed} \implies$
 $\text{denied1} \subseteq \text{setbydecision-all TYPE('pkt-ext) iface } rs1 \text{ FinalDeny} \implies$
 $\text{no-spoofing-algorithm iface } ipassmt \text{ } rs2 \text{ allowed denied1} \implies$
 $\text{nospoof TYPE('pkt-ext) iface } ipassmt (rs1 @ rs2)$
 $\langle \text{proof} \rangle$

definition $\text{no-spoofing-iface} :: \text{iface} \Rightarrow 'i::\text{len} \text{ ipassignment} \Rightarrow 'i \text{ common-primitive rule list} \Rightarrow \text{bool}$ **where**

$\text{no-spoofing-iface } \text{iface } ipassmt \text{ } rs \equiv \text{no-spoofing-algorithm } \text{iface } ipassmt \text{ } rs \{ \} \{ \}$

lemma[code]: $\text{no-spoofing-iface } \text{iface } ipassmt \text{ } rs =$

$\text{no-spoofing-algorithm-executable } \text{iface } ipassmt \text{ } rs \text{ Empty-WordInterval Empty-WordInterval}$

$\langle \text{proof} \rangle \text{ corollary no-spoofing-algorithm-sound: simple-ruleset } rs \implies \forall r \in \text{set } rs. \text{normalized-nnf-match } (\text{get-match } r) \implies$

$\text{no-spoofing-iface } \text{iface } ipassmt \text{ } rs \implies \text{nospoof TYPE('pkt-ext) iface } ipassmt$

rs

$\langle \text{proof} \rangle$

The *nospoof* definition used throughout the proofs corresponds to checking *no-spoofing* for all interfaces

private lemma *nospoof*: $\text{simple-ruleset } rs \implies (\forall \text{iface} \in \text{dom } ipassmt. \text{nospoof TYPE('pkt-ext) iface } ipassmt \text{ } rs) \iff \text{no-spoofing TYPE('pkt-ext) ipassmt } rs$

$\langle \text{proof} \rangle$

theorem *no-spoofing-iface: simple-ruleset* $rs \implies \forall r \in \text{set } rs. \text{normalized-nnf-match}$
(get-match r) \implies
 $\forall \text{iface} \in \text{dom } \text{ipassmt}. \text{no-spoofing-iface } \text{iface } \text{ipassmt } rs \implies \text{no-spoofing}$
TYPE('pkt-ext) ipassmt rs
<proof>

Examples

Example 1: Ruleset: Accept all non-spoofed packets, drop rest.

lemma *no-spoofing-iface*
(Iface "eth0")
 $[\text{Iface "eth0"} \mapsto [(\text{ipv4addr-of-dotdecimal } (192,168,0,0), 24)]]$
 $[\text{Rule } (\text{MatchAnd } (\text{Match } (\text{Src } (\text{IpAddrNetmask } (\text{ipv4addr-of-dotdecimal } (192,168,0,0)) 24))) (\text{Match } (\text{Iiface } (\text{Iface "eth0"})))) \text{action.Accept},$
 $\text{Rule MatchAny action.Drop}]$ *<proof>*

lemma *no-spoofing TYPE('pkt-ext)*
 $[\text{Iface "eth0"} \mapsto [(\text{ipv4addr-of-dotdecimal } (192,168,0,0), 24)]]$
 $[\text{Rule } (\text{MatchAnd } (\text{Match } (\text{Src } (\text{IpAddrNetmask } (\text{ipv4addr-of-dotdecimal } (192,168,0,0)) 24))) (\text{Match } (\text{Iiface } (\text{Iface "eth0"})))) \text{action.Accept},$
 $\text{Rule MatchAny action.Drop}]$
<proof>

Example 2: Ruleset: Drop packets from a spoofed IP range, allow rest. Handles negated interfaces correctly.

lemma *no-spoofing TYPE('pkt-ext)*
 $[\text{Iface "eth0"} \mapsto [(\text{ipv4addr-of-dotdecimal } (192,168,0,0), 24)]]$
 $[\text{Rule } (\text{MatchAnd } (\text{Match } (\text{Iiface } (\text{Iface "wlan+"}))) (\text{Match } (\text{Extra "no idea what this is"}))) \text{action.Accept},$ — not interesting for spoofing
 $\text{Rule } (\text{MatchNot } (\text{Match } (\text{Iiface } (\text{Iface "eth0+"})))) \text{action.Accept},$ — not interesting for spoofing
 $\text{Rule } (\text{MatchAnd } (\text{MatchNot } (\text{Match } (\text{Src } (\text{IpAddrNetmask } (\text{ipv4addr-of-dotdecimal } (192,168,0,0)) 24)))) (\text{Match } (\text{Iiface } (\text{Iface "eth0"})))) \text{action.Drop},$ — spoof-protect here
 $\text{Rule MatchAny action.Accept}]$
<proof>

Example 3: Accidentally, matching on wlan+, spoofed packets for eth0 are allowed. First, we prove that there actually is no spoofing protection. Then we show that our algorithm finds out.

lemma \neg *no-spoofing TYPE('pkt-ext)*
 $[\text{Iface "eth0"} \mapsto [(\text{ipv4addr-of-dotdecimal } (192,168,0,0), 24)]]$
 $[\text{Rule } (\text{MatchNot } (\text{Match } (\text{Iiface } (\text{Iface "wlan+"})))) \text{action.Accept},$ —
 accidentally allow everything for eth0
 $\text{Rule } (\text{MatchAnd } (\text{MatchNot } (\text{Match } (\text{Src } (\text{IpAddrNetmask } (\text{ipv4addr-of-dotdecimal } (192,168,0,0)) 24)))) (\text{Match } (\text{Iiface } (\text{Iface "eth0"})))) \text{action.Drop},$
 $\text{Rule MatchAny action.Accept}]$

<proof>

lemma \neg *no-spoofing-iface*

(*Iface "eth0"*)

[*Iface "eth0"* \mapsto [(*ipv4addr-of-dotdecimal* (192,168,0,0), 24)]]

[*Rule* (*MatchNot* (*Match* (*Iiface* (*Iface "wlan+"*)))) *action.Accept*, —

accidentally allow everything for eth0

Rule (*MatchAnd* (*MatchNot* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (192,168,0,0) 24)))) (*Match* (*Iiface* (*Iface "eth0"*)))) *action.Drop*,

Rule MatchAny action.Accept]

<proof>

Example 4: Ruleset: Drop packets coming from the wrong interface, allow the rest. Warning: this does not prevent spoofing for eth0! Explanation: someone on eth0 can send a packet e.g. with source IP 8.8.8.8 The ruleset only prevents spoofing of 192.168.0.0/24 for other interfaces

lemma \neg *no-spoofing TYPE('pkt-ext)* [*Iface "eth0"* \mapsto [(*ipv4addr-of-dotdecimal* (192,168,0,0), 24)]]

[*Rule* (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (192,168,0,0) 24)))) (*MatchNot* (*Match* (*Iiface* (*Iface "eth0"*)))) *action.Drop*,

Rule MatchAny action.Accept]

<proof>

Our algorithm detects it.

lemma \neg *no-spoofing-iface*

(*Iface "eth0"*)

[*Iface "eth0"* \mapsto [(*ipv4addr-of-dotdecimal* (192,168,0,0), 24)]]

[*Rule* (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (192,168,0,0) 24)))) (*MatchNot* (*Match* (*Iiface* (*Iface "eth0"*)))) *action.Drop*,

Rule MatchAny action.Accept] *<proof>*

Example 5: Spoofing protection but the algorithm fails. The algorithm *no-spoofing-iface* is only sound, not complete. The ruleset first drops spoofed packets for TCP and then drops spoofed packets for \neg TCP. The algorithm cannot detect that $TCP \cup \neg TCP$ together will match all spoofed packets.

lemma *no-spoofing TYPE('pkt-ext)* [*Iface "eth0"* \mapsto [(*ipv4addr-of-dotdecimal* (192,168,0,0), 24)]]

[*Rule* (*MatchAnd* (*MatchNot* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (192,168,0,0) 24))))

(*MatchAnd* (*Match* (*Iiface* (*Iface "eth0"*))

(*Match* (*Prot* (*Proto TCP*)))) *action.Drop*,

Rule (*MatchAnd* (*MatchNot* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (192,168,0,0) 24))))

(*MatchAnd* (*Match* (*Iiface* (*Iface "eth0"*))

(*MatchNot* (*Match* (*Prot* (*Proto TCP*)))) *action.Drop*,

Rule MatchAny action.Accept] (**is** *no-spoofing TYPE('pkt-ext)* *?ipassmt*

?rs)

<proof>

Spoofing protection but the algorithm cannot certify spoofing protection.

```

lemma  $\neg$  no-spoofing-iface
  (Iface "eth0")
  [Iface "eth0"  $\mapsto$  [(ipv4addr-of-dotdecimal (192,168,0,0), 24)]]
  [Rule (MatchAnd (MatchNot (Match (Src (IpAddrNetmask (ipv4addr-of-dotdecimal
    (192,168,0,0) 24))))
    (MatchAnd (Match (Iiface (Iface "eth0")))
      (Match (Prot (Proto TCP)))))) action.Drop,
    Rule (MatchAnd (MatchNot (Match (Src (IpAddrNetmask (ipv4addr-of-dotdecimal
      (192,168,0,0) 24))))
      (MatchAnd (Match (Iiface (Iface "eth0")))
        (MatchNot (Match (Prot (Proto TCP)))))) action.Drop,
      Rule MatchAny action.Accept] <proof>

```

end

```

lemma no-spoofing-iface (Iface "eth1.1011")
  ([Iface "eth1.1011"  $\mapsto$  [(ipv4addr-of-dotdecimal (131,159,14,0),
    24)]]:: 32 ipassignment)
  [Rule (MatchNot (Match (Iiface (Iface "eth1.1011+")))) action.Accept,
    Rule (MatchAnd (MatchNot (Match (Src (IpAddrNetmask (ipv4addr-of-dotdecimal
      (131,159,14,0) 24)))) (Match (Iiface (Iface "eth1.1011")))) action.Drop,
      Rule MatchAny action.Accept] <proof>

```

We only check accepted packets. If there is no default rule (this will never happen if parsed from iptables!), the result is unfinished.

```

lemma no-spoofing-iface (Iface "eth1.1011")
  ([Iface "eth1.1011"  $\mapsto$  [(ipv4addr-of-dotdecimal (131,159,14,0),
    24)]]:: 32 ipassignment)
  [Rule (Match (Src (IpAddrNetmask (ipv4addr-of-dotdecimal (127, 0, 0, 0) 8)))
    Drop] <proof>

```

end

```

theory Common-Primitive-toString
imports Simple-Firewall.Primitives-toString
  Common-Primitive-Matcher

```

begin

33 Firewall toString Functions

```

fun ipt-ipv4range-toString :: 32 ipt-irange  $\Rightarrow$  string where
  ipt-ipv4range-toString (IpAddr ip) = ipv4addr-toString ip |
  ipt-ipv4range-toString (IpAddrNetmask ip n) = ipv4addr-toString ip@"@"string-of-nat
  n |
  ipt-ipv4range-toString (IpAddrRange ip1 ip2) = ipv4addr-toString ip1@"-"@ipv4addr-toString
  ip2

```

```

fun ipt-ipv6range-toString :: 128 ipt-irange  $\Rightarrow$  string where

```

```

    ipt-ipv6range-toString (IpAddr ip) = ipv6addr-toString ip |
    ipt-ipv6range-toString (IpAddrNetmask ip n) = ipv6addr-toString ip@"@"string-of-nat
n |
    ipt-ipv6range-toString (IpAddrRange ip1 ip2) = ipv6addr-toString ip1@"-"@ipv6addr-toString
ip2

```

definition *ipv4addr-wordinterval-pretty-toString* :: 32 wordinterval ⇒ string **where**
ipv4addr-wordinterval-pretty-toString wi = list-toString ipt-ipv4range-toString (wi-to-ipt-iprange wi)

lemma *ipv4addr-wordinterval-pretty-toString*
(RangeUnion (RangeUnion (WordInterval 0x7F000000 0x7FFFFFFF) (WordInterval 0x1020304 0x1020306)) (WordInterval 0x8080808 0x8080808)) = "[127.0.0.0/8, 1.2.3.4-1.2.3.6, 8.8.8.8]" <proof>

fun *action-toString* :: action ⇒ string **where**
action-toString action.Accept = "-j ACCEPT" |
action-toString action.Drop = "-j DROP" |
action-toString action.Reject = "-j REJECT" |
action-toString (action.Call target) = "-j "@target@" (call)" |
action-toString (action.Goto target) = "-g "@target" |
action-toString action.Empty = "" |
action-toString action.Log = "-j LOG" |
action-toString action.Return = "-j RETURN" |
action-toString action.Unknown = "!!!!!!!!!! UNKNOWN !!!!!!!!!!!"

fun *common-primitive-toString* :: ('i::len word ⇒ string) ⇒ 'i common-primitive ⇒ string **where**
common-primitive-toString ipToStr (Src (IpAddr ip)) = "-s "@ipToStr ip |
common-primitive-toString ipToStr (Dst (IpAddr ip)) = "-d "@ipToStr ip |
common-primitive-toString ipToStr (Src (IpAddrNetmask ip n)) = "-s "@ipToStr ip@"@"string-of-nat n |
common-primitive-toString ipToStr (Dst (IpAddrNetmask ip n)) = "-d "@ipToStr ip@"@"string-of-nat n |
common-primitive-toString ipToStr (Src (IpAddrRange ip1 ip2)) = "-m iprange --src-range "@ipToStr ip1@"-"@ipToStr ip2 |
common-primitive-toString ipToStr (Dst (IpAddrRange ip1 ip2)) = "-m iprange --dst-range "@ipToStr ip1@"-"@ipToStr ip2 |
common-primitive-toString - (Iface ifce) = iface-toString "-i " ifce |
common-primitive-toString - (Oiface ifce) = iface-toString "-o " ifce |
common-primitive-toString - (Prot prot) = "-p "@protocol-toString prot |
common-primitive-toString - (Src-Ports (L4Ports prot pts)) = "-m "@primitive-protocol-toString prot@" --spts " @ list-toString (ports-toString "'") pts |
common-primitive-toString - (Dst-Ports (L4Ports prot pts)) = "-m "@primitive-protocol-toString prot@" --dpts " @ list-toString (ports-toString "'") pts |

```

    common-primitive-toString - (MultiportPorts (L4Ports prot pts)) = "-p "@primitive-protocol-toString
    prot@" -m multiport --ports " @ list-toString (ports-toString ""') pts |
    common-primitive-toString - (CT-State S) = "-m state --state "@ctstate-set-toString
    S |
    common-primitive-toString - (L4-Flags (TCP-Flags c m)) = "--tcp-flags "@ipt-tcp-flags-toString
    c@" "@ipt-tcp-flags-toString m |
    common-primitive-toString - (Extra e) = "~~"@e@"~~"

```

definition *common-primitive-ipv4-toString* :: 32 *common-primitive* \Rightarrow *string* **where**
common-primitive-ipv4-toString \equiv *common-primitive-toString* *ipv4addr-toString*

definition *common-primitive-ipv6-toString* :: 128 *common-primitive* \Rightarrow *string* **where**
common-primitive-ipv6-toString \equiv *common-primitive-toString* *ipv6addr-toString*

```

fun common-primitive-match-expr-toString
  :: ('i common-primitive  $\Rightarrow$  string)  $\Rightarrow$  'i common-primitive match-expr  $\Rightarrow$  string
where
  common-primitive-match-expr-toString toStr MatchAny = "" |
  common-primitive-match-expr-toString toStr (Match m) = toStr m |
  common-primitive-match-expr-toString toStr (MatchAnd m1 m2) =
    common-primitive-match-expr-toString toStr m1 @"" @ common-primitive-match-expr-toString
    toStr m2 |
  common-primitive-match-expr-toString toStr (MatchNot (Match m)) = "! "@toStr
  m |
  common-primitive-match-expr-toString toStr (MatchNot m) = "NOT (" @ common-primitive-match-expr-toSt
  toStr m @'"')

```

definition *common-primitive-match-expr-ipv4-toString* :: 32 *common-primitive match-expr*
 \Rightarrow *string* **where**
common-primitive-match-expr-ipv4-toString \equiv *common-primitive-match-expr-toString*
common-primitive-ipv4-toString

definition *common-primitive-match-expr-ipv6-toString* :: 128 *common-primitive*
match-expr \Rightarrow *string* **where**
common-primitive-match-expr-ipv6-toString \equiv *common-primitive-match-expr-toString*
common-primitive-ipv6-toString

```

fun common-primitive-rule-toString :: 32 common-primitive rule  $\Rightarrow$  string where
  common-primitive-rule-toString (Rule m a) = common-primitive-match-expr-ipv4-toString
  m @"" "@action-toString a

```

end

34 Routing and IP Assignments

theory *Routing-IpAssmt*

```

imports Ipassmt
         Routing.Routing-Table
begin
context
begin

```

34.1 Routing IP Assignment

Up to now, the definitions were all still on word intervals because those are much more convenient to work with.

definition *routing-ipassmt* :: 'i::len *routing-rule list* \Rightarrow (*iface* \times ('i *word* \times *nat*) *list*) *list*

where

routing-ipassmt *rt* \equiv *map* (*apfst* *Iface* \circ *apsnd* *cidr-split*) (*routing-ipassmt-wi* *rt*)

private lemma *ipcidr-union-cidr-split[simp]*: *ipcidr-union-set* (*set* (*cidr-split* *x*)) = *wordinterval-to-set* *x*

<proof> **lemma** *map-of-map-Iface*: *map-of* (*map* ($\lambda x. (Iface (fst\ x), f (snd\ x))$)) *xs* (*Iface* *ifce*) =

map-option *f* ((*map-of* *xs*) *ifce*)

<proof>

lemma *routing-ipassmt-wi* ([::32 *prefix-routing*) = [(*output-iface* (*routing-action* (*undefined* :: 32 *routing-rule*)), *WordInterval* 0 0xFFFFFFFF)]

<proof>

lemma *routing-ipassmt*:

valid-prefixes *rt* \implies

output-iface (*routing-table-semantics* *rt* (*p-dst* *p*)) = *p-iface* *p* \implies

\exists *p-ips*. *map-of* (*routing-ipassmt* *rt*) (*Iface* (*p-iface* *p*)) = *Some* *p-ips* \wedge *p-dst* *p* \in *ipcidr-union-set* (*set* *p-ips*)

<proof>

lemma *routing-ipassmt-ipassmt-sanity-disjoint*: *valid-prefixes* (*rt*::('i::len) *prefix-routing*) \implies

ipassmt-sanity-disjoint (*map-of* (*routing-ipassmt* *rt*))

<proof>

lemma *routing-ipassmt-distinct*: *distinct* (*map* *fst* (*routing-ipassmt* *rtbl*))

<proof>

end

end

theory *Output-Interface-Replace*

imports

Ipassmt

Routing-IpAssmt

Common-Primitive-toString
begin

35 Replacing output interfaces by their IP ranges according to Routing

Copy of `Interface_Replace.thy`

definition *ipassmt-iface-replace-dstip-mexpr*

$:: 'i::len\ ipassignment \Rightarrow iface \Rightarrow 'i\ common\ primitive\ match\ expr\ \mathbf{where}$
 $ipassmt\ iface\ replace\ dstip\ mexpr\ ipassmt\ ifce \equiv case\ ipassmt\ ifce\ of$
 $None \Rightarrow Match\ (OIface\ ifce)$
 $| Some\ ips \Rightarrow (match\ list\ to\ match\ expr\ (map\ (Match\ \circ\ Dst)\ (map\ (uncurry\ IpAddrNetmask)\ ips)))$

lemma *matches-ipassmt-iface-replace-dstip-mexpr:*

$matches\ (common\ matcher,\ \alpha)\ (ipassmt\ iface\ replace\ dstip\ mexpr\ ipassmt\ ifce)$
 $a\ p \longleftrightarrow (case\ ipassmt\ ifce\ of$
 $None \Rightarrow match\ iface\ ifce\ (p\ oiface\ p)$
 $| Some\ ips \Rightarrow p\ dst\ p \in\ ipcidr\ union\ set\ (set\ ips)$
 $)$
 $\langle proof \rangle$

fun *oiface-rewrite*

$:: 'i::len\ ipassignment \Rightarrow 'i\ common\ primitive\ match\ expr \Rightarrow 'i\ common\ primitive\ match\ expr$

where

$oiface\ rewrite - MatchAny = MatchAny |$
 $oiface\ rewrite\ ipassmt\ (Match\ (OIface\ ifce)) = ipassmt\ iface\ replace\ dstip\ mexpr\ ipassmt\ ifce |$
 $oiface\ rewrite - (Match\ a) = Match\ a |$
 $oiface\ rewrite\ ipassmt\ (MatchNot\ m) = MatchNot\ (oiface\ rewrite\ ipassmt\ m) |$
 $oiface\ rewrite\ ipassmt\ (MatchAnd\ m1\ m2) = MatchAnd\ (oiface\ rewrite\ ipassmt\ m1)\ (oiface\ rewrite\ ipassmt\ m2)$

context

begin

private lemma *oiface-rewrite-matches-Primitive:*

$matches\ (common\ matcher,\ \alpha)\ (MatchNot\ (oiface\ rewrite\ ipassmt\ (Match\ x)))\ a\ p = matches\ (common\ matcher,\ \alpha)\ (MatchNot\ (Match\ x))\ a\ p \longleftrightarrow$
 $matches\ (common\ matcher,\ \alpha)\ (oiface\ rewrite\ ipassmt\ (Match\ x))\ a\ p =$
 $matches\ (common\ matcher,\ \alpha)\ (Match\ x)\ a\ p$
 $\langle proof \rangle$

lemma *ipassmt-disjoint-matcheq-ifce-dstip:*

assumes *ipassmt-nowild: ipassmt-sanity-nowildcards ipassmt*

```

and ipassmt-disjoint: ipassmt-sanity-disjoint ipassmt
and ifce: ipassmt ifce = Some i-ips
and p-ifce: ipassmt (Iface (p-oiface p)) = Some p-ips  $\wedge$  p-dst p  $\in$ 
ipcidr-union-set (set p-ips)
shows match-iface ifce (p-oiface p)  $\longleftrightarrow$  p-dst p  $\in$  ipcidr-union-set (set
i-ips)
  <proof> lemma matches-ipassmt-iface-replace-dstip-mexpr-case-Iface:
    fixes ifce::iface
    assumes ipassmt-sanity-nowildcards ipassmt
    and ipassmt-sanity-disjoint ipassmt
    and ipassmt (Iface (p-oiface p)) = Some p-ips  $\wedge$  p-dst p  $\in$  ipcidr-union-set
(set p-ips)
    shows matches (common-matcher,  $\alpha$ ) (ipassmt-iface-replace-dstip-mexpr
ipassmt ifce) a p  $\longleftrightarrow$ 
      matches (common-matcher,  $\alpha$ ) (Match (OIface ifce)) a p
  <proof>

```

```

lemma matches-oiface-rewrite-ipassmt:
  normalized-nnf-match m  $\implies$  ipassmt-sanity-nowildcards ipassmt  $\implies$  ipassmt-sanity-disjoint
ipassmt  $\implies$ 
  ( $\exists$  p-ips. ipassmt (Iface (p-oiface p)) = Some p-ips  $\wedge$  p-dst p  $\in$  ipcidr-union-set
(set p-ips))  $\implies$ 
  matches (common-matcher,  $\alpha$ ) (oiface-rewrite ipassmt m) a p  $\longleftrightarrow$  matches
(common-matcher,  $\alpha$ ) m a p
  <proof>

```

```

lemma matches-oiface-rewrite:
  normalized-nnf-match m  $\implies$  ipassmt-sanity-nowildcards ipassmt — TODO:
  check?  $\implies$ 
  correct-routing rt  $\implies$ 
  ipassmt = map-of (routing-ipassmt rt)  $\implies$ 
  output-iface (routing-table-semantics rt (p-dst p)) = p-oiface p  $\implies$ 
  matches (common-matcher,  $\alpha$ ) (oiface-rewrite ipassmt m) a p  $\longleftrightarrow$  matches
(common-matcher,  $\alpha$ ) m a p
  <proof>
end

```

```

lemma oiface-rewrite-preserves-nodisc:
   $\forall a. \neg disc (Dst a)  $\implies$  \neg has-disc disc m  $\implies$  \neg has-disc disc (oiface-rewrite$ 
ipassmt m)
  <proof>

```

```

end
theory Interface-Replace
imports
  No-Spoof

```

```

    Common-Primitive-toString
    Output-Interface-Replace
begin

```

36 Trying to connect inbound interfaces by their IP ranges

36.1 Constraining Interfaces

We keep the match on the interface but add the corresponding IP address range.

```

definition ipassmt-iface-constrain-srcip-mexpr
  :: 'i::len ipassignment ⇒ iface ⇒ 'i common-primitive match-expr
where
  ipassmt-iface-constrain-srcip-mexpr ipassmt ifce = (case ipassmt ifce of
    None ⇒ Match (Iiface ifce)
  | Some ips ⇒ MatchAnd
    (Match (Iiface ifce))
    (match-list-to-match-expr (map (Match ∘ Src) (map (uncurry IpAddr-
Netmask) ips)))
  )

```

```

lemma matches-ipassmt-iface-constrain-srcip-mexpr:
  matches (common-matcher, α) (ipassmt-iface-constrain-srcip-mexpr ipassmt
ifce) a p ↔
  (case ipassmt ifce of
    None ⇒ match-iface ifce (p-iiface p)
  | Some ips ⇒ match-iface ifce (p-iiface p) ∧ p-src p ∈ ipcidr-union-set (set
ips)
  )
⟨proof⟩

```

```

fun iiface-constrain :: 'i::len ipassignment ⇒ 'i common-primitive match-expr ⇒
'i common-primitive match-expr where
  iiface-constrain - MatchAny = MatchAny |
  iiface-constrain ipassmt (Match (Iiface ifce)) = ipassmt-iface-constrain-srcip-mexpr
ipassmt ifce |
  iiface-constrain ipassmt (Match a) = Match a |
  iiface-constrain ipassmt (MatchNot m) = MatchNot (iiface-constrain ipassmt m)
|
  iiface-constrain ipassmt (MatchAnd m1 m2) = MatchAnd (iiface-constrain ipassmt
m1) (iiface-constrain ipassmt m2)

```

```

context
begin

```


private lemma *iiface-constrain-matches-Primitive:*
 $\text{matches } (\text{common-matcher}, \alpha) (\text{MatchNot } (\text{iiface-constrain } \text{ipassmt } (\text{Match } x))) a p = \text{matches } (\text{common-matcher}, \alpha) (\text{MatchNot } (\text{Match } x)) a p \longleftrightarrow$
 $\text{matches } (\text{common-matcher}, \alpha) (\text{iiface-constrain } \text{ipassmt } (\text{Match } x)) a p$
 $= \text{matches } (\text{common-matcher}, \alpha) (\text{Match } x) a p$
 $\langle \text{proof} \rangle$ **lemma** *matches-ipassmt-iiface-constrain-srcip-mexpr-case-Iiface:*
fixes *iface::iface*
assumes *ipassmt-sanity-nowildcards ipassmt*
and $\bigwedge \text{ips. } \text{ipassmt } (\text{Iiface } (p\text{-iiface } p)) = \text{Some } \text{ips} \implies p\text{-src } p \in \text{ipcidr-union-set } (\text{set } \text{ips})$
shows $\text{matches } (\text{common-matcher}, \alpha) (\text{ipassmt-iiface-constrain-srcip-mexpr } \text{ipassmt } \text{iface}) a p \longleftrightarrow$
 $\text{matches } (\text{common-matcher}, \alpha) (\text{Match } (\text{Iiface } \text{iface})) a p$
 $\langle \text{proof} \rangle$

lemma *matches-iiface-constrain:*
 $\text{normalized-nnf-match } m \implies \text{ipassmt-sanity-nowildcards } \text{ipassmt} \implies$
 $(\bigwedge \text{ips. } \text{ipassmt } (\text{Iiface } (p\text{-iiface } p)) = \text{Some } \text{ips} \implies p\text{-src } p \in \text{ipcidr-union-set } (\text{set } \text{ips})) \implies$
 $\text{matches } (\text{common-matcher}, \alpha) (\text{iiface-constrain } \text{ipassmt } m) a p \longleftrightarrow \text{matches } (\text{common-matcher}, \alpha) m a p$
 $\langle \text{proof} \rangle$
end

36.2 Sanity checking the assumption

lemma $(\exists \text{ips. } \text{ipassmt } (\text{Iiface } (p\text{-iiface } p)) = \text{Some } \text{ips} \wedge p\text{-src } p \in \text{ipcidr-union-set } (\text{set } \text{ips})) \implies$
 $(\text{case } \text{ipassmt } (\text{Iiface } (p\text{-iiface } p)) \text{ of } \text{Some } \text{ips} \Rightarrow p\text{-src } p \in \text{ipcidr-union-set } (\text{set } \text{ips}))$
 $(\text{case } \text{ipassmt } (\text{Iiface } (p\text{-iiface } p)) \text{ of } \text{Some } \text{ips} \Rightarrow p\text{-src } p \in \text{ipcidr-union-set } (\text{set } \text{ips})) \implies$
 $(\bigwedge \text{ips. } \text{ipassmt } (\text{Iiface } (p\text{-iiface } p)) = \text{Some } \text{ips} \implies p\text{-src } p \in \text{ipcidr-union-set } (\text{set } \text{ips}))$
 $\langle \text{proof} \rangle$

Sanity check: If we assume that there are no spoofed packets, spoofing protection is trivially fulfilled.

lemma $\forall p:: ('i::\text{len}, 'pkt\text{-ext}) \text{tagged-packet-scheme.}$
 $\text{Iiface } (p\text{-iiface } p) \in \text{dom } \text{ipassmt} \longrightarrow p\text{-src } p \in \text{ipcidr-union-set } (\text{set } (\text{the } (\text{ipassmt } (\text{Iiface } (p\text{-iiface } p)))))) \implies$
 $\text{no-spoofing } \text{TYPE}('pkt\text{-ext}) \text{ipassmt } rs$
 $\langle \text{proof} \rangle$

Sanity check: If the firewall features spoofing protection and we look at a packet which was allowed by the firewall. Then the packet's src ip must be according to ipassmt. (case Some) We don't case about packets from an interface which are not defined in ipassmt. (case None)

lemma

fixes $p :: ('i::len, 'pkt-ext) \text{ tagged-packet-scheme}$
shows $\text{no-spoofing TYPE('pkt-ext) ipassmt rs} \implies$
 $(\text{common-matcher, in-doubt-allow}) \text{p} \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow}$
 \implies
 $\text{case ipassmt (Iface (p-iiface p)) of Some ips} \Rightarrow p\text{-src } p \in \text{ipcidr-union-set}$
 $(\text{set ips}) \mid \text{None} \Rightarrow \text{True}$
 $\langle \text{proof} \rangle$

36.3 Replacing Interfaces Completely

This is a stricter, true rewriting since it removes the interface match completely. However, it requires *ipassmt-sanity-disjoint*

thm *ipassmt-sanity-disjoint-def*

definition *ipassmt-iface-replace-srcip-mexpr*

$:: 'i::len \text{ ipassignment} \Rightarrow \text{iface} \Rightarrow 'i \text{ common-primitive match-expr}$ **where**
 $\text{ipassmt-iface-replace-srcip-mexpr ipassmt ifce} \equiv \text{case ipassmt ifce of}$
 $\text{None} \Rightarrow \text{Match (Iface ifce)}$
 $\mid \text{Some ips} \Rightarrow (\text{match-list-to-match-expr } (\text{map } (\text{Match} \circ \text{Src}) (\text{map } (\text{uncurry}$
 $\text{IpAddrNetmask}) \text{ ips})))$

lemma *matches-ipassmt-iface-replace-srcip-mexpr:*

$\text{matches } (\text{common-matcher}, \alpha) (\text{ipassmt-iface-replace-srcip-mexpr ipassmt ifce})$
 $a \text{ p} \longleftrightarrow (\text{case ipassmt ifce of}$
 $\text{None} \Rightarrow \text{match-iface ifce (p-iiface p)}$
 $\mid \text{Some ips} \Rightarrow p\text{-src } p \in \text{ipcidr-union-set } (\text{set ips})$
 $)$
 $\langle \text{proof} \rangle$

fun *iiface-rewrite*

$:: 'i::len \text{ ipassignment} \Rightarrow 'i \text{ common-primitive match-expr} \Rightarrow 'i \text{ common-primitive}$
 match-expr

where

$\text{iiface-rewrite - MatchAny} = \text{MatchAny} \mid$
 $\text{iiface-rewrite ipassmt (Match (Iface ifce))} = \text{ipassmt-iface-replace-srcip-mexpr}$
 $\text{ipassmt ifce} \mid$
 $\text{iiface-rewrite ipassmt (Match a)} = \text{Match a} \mid$
 $\text{iiface-rewrite ipassmt (MatchNot m)} = \text{MatchNot (iiface-rewrite ipassmt m)} \mid$
 $\text{iiface-rewrite ipassmt (MatchAnd m1 m2)} = \text{MatchAnd (iiface-rewrite ipassmt}$
 $m1) (\text{iiface-rewrite ipassmt } m2)$

context

begin

private lemma *iiface-rewrite-matches-Primitive:*

```

    matches (common-matcher,  $\alpha$ ) (MatchNot (iface-rewrite ipassmt (Match
x))) a p = matches (common-matcher,  $\alpha$ ) (MatchNot (Match x)) a p  $\longleftrightarrow$ 
    matches (common-matcher,  $\alpha$ ) (iface-rewrite ipassmt (Match x)) a p =
matches (common-matcher,  $\alpha$ ) (Match x) a p
  <proof> lemma matches-ipassmt-iface-replace-srcip-mexpr-case-Iface:
    fixes ifce::iface
    assumes ipassmt-sanity-nowildcards ipassmt
    and ipassmt-sanity-disjoint ipassmt
    and ipassmt (Iface (p-iface p)) = Some p-ips  $\wedge$  p-src p  $\in$  ipcidr-union-set
(set p-ips)
    shows matches (common-matcher,  $\alpha$ ) (ipassmt-iface-replace-srcip-mexpr
ipassmt ifce) a p  $\longleftrightarrow$ 
    matches (common-matcher,  $\alpha$ ) (Match (Iiface ifce)) a p
  <proof>

```

```

lemma matches-iface-rewrite:
  normalized-nnf-match m  $\implies$  ipassmt-sanity-nowildcards ipassmt  $\implies$  ipassmt-sanity-disjoint
ipassmt  $\implies$ 
  ( $\exists$  p-ips. ipassmt (Iface (p-iface p)) = Some p-ips  $\wedge$  p-src p  $\in$  ipcidr-union-set
(set p-ips))  $\implies$ 
  matches (common-matcher,  $\alpha$ ) (iface-rewrite ipassmt m) a p  $\longleftrightarrow$  matches
(common-matcher,  $\alpha$ ) m a p
  <proof>

```

end

Finally, we show that *ipassmt-sanity-disjoint* is really needed.

```

lemma iface-replace-needs-ipassmt-disjoint:
  assumes ipassmt-sanity-nowildcards ipassmt
  and iface-replace:  $\bigwedge$  ifce p:: 'i::len tagged-packet.
    (matches (common-matcher,  $\alpha$ ) (ipassmt-iface-replace-srcip-mexpr ipassmt
ifce) a p  $\longleftrightarrow$  matches (common-matcher,  $\alpha$ ) (Match (Iiface ifce)) a p)
  shows ipassmt-sanity-disjoint ipassmt
  <proof>

```

end

```

theory Optimizing
imports Semantics-Ternary
begin

```

37 Optimizing

37.1 Removing Shadowed Rules

Note: there is no executable code for `rmshadow` at the moment

Assumes: *simple-ruleset*

```

fun rmshadow :: ('a, 'p) match-tac  $\Rightarrow$  'a rule list  $\Rightarrow$  'p set  $\Rightarrow$  'a rule list where
  rmshadow - [] - = [] |
  rmshadow  $\gamma$  ((Rule m a)#rs) P = (if ( $\forall p \in P. \neg$  matches  $\gamma$  m a p)
    then
      rmshadow  $\gamma$  rs P
    else
      (Rule m a) # (rmshadow  $\gamma$  rs {p  $\in$  P.  $\neg$  matches  $\gamma$  m a p}))

```

37.1.1 Soundness

lemma *rmshadow-sound*:
 $simple\text{-ruleset } rs \Longrightarrow p \in P \Longrightarrow approximating\text{-bigstep-fun } \gamma p (rmshadow \gamma rs P) = approximating\text{-bigstep-fun } \gamma p rs$
 <proof>

37.2 Removing rules which cannot apply

```

fun rmMatchFalse :: 'a rule list  $\Rightarrow$  'a rule list where
  rmMatchFalse [] = [] |
  rmMatchFalse ((Rule (MatchNot MatchAny) -)#rs) = rmMatchFalse rs |
  rmMatchFalse (r#rs) = r # rmMatchFalse rs

```

lemma *rmMatchFalse-correct*: $approximating\text{-bigstep-fun } \gamma p (rmMatchFalse rs) s = approximating\text{-bigstep-fun } \gamma p rs s$
 <proof>

We can stop after a default rule (a rule which matches anything) is observed.

```

fun cut-off-after-match-any :: 'a rule list  $\Rightarrow$  'a rule list where
  cut-off-after-match-any [] = [] |
  cut-off-after-match-any (Rule m a # rs) =
    (if m = MatchAny  $\wedge$  (a = Accept  $\vee$  a = Drop  $\vee$  a = Reject)
     then [Rule m a] else Rule m a # cut-off-after-match-any rs)

```

lemma *cut-off-after-match-any*:
 $approximating\text{-bigstep-fun } \gamma p (cut\text{-off-after-match-any } rs) s = approximating\text{-bigstep-fun } \gamma p rs s$
 <proof>

lemma *cut-off-after-match-any-simplers*: $simple\text{-ruleset } rs \Longrightarrow simple\text{-ruleset } (cut\text{-off-after-match-any } rs)$
 <proof>

lemma *cut-off-after-match-any-preserve-matches*:
 $\forall r \in set\ rs. P (get\text{-match } r) \Longrightarrow \forall r \in set (cut\text{-off-after-match-any } rs). P (get\text{-match } r)$
 <proof>

end

38 Optimizing and Normalizing Primitives

```

theory Transform
imports Common-Primitive-Lemmas
        ../Semantics-Ternary/Semantics-Ternary
        ../Semantics-Ternary/Negation-Type-Matching
        Ports-Normalize
        IpAddresses-Normalize
        Interfaces-Normalize
        Protocols-Normalize
        ../Common/Remdups-Rev
        Interface-Replace
        ../Semantics-Ternary/Optimizing

```

begin

This transform theory plugs a lot of stuff together. We perform several normalization and optimization steps on complete firewall rulesets. We show that it preserves the semantics and also, that structural properties are preserved. For example, if you normalize interfaces and afterwards normalize protocols, the interfaces remain normalized and no new interfaces are added when doing the protocol normalization.

definition *compress-normalize-besteffort*
 $:: 'i::len \text{ common-primitive match-expr} \Rightarrow 'i \text{ common-primitive match-expr option}$
where

$$\text{compress-normalize-besteffort } m \equiv \text{compress-normalize-primitive-monad} \\
[\text{compress-normalize-protocols}, \\
\text{compress-normalize-input-interfaces}, \\
\text{compress-normalize-output-interfaces}] m$$

context begin

private lemma *compress-normalize-besteffort-normalized:*

$$f \in \text{set} [\text{compress-normalize-protocols}, \\
\text{compress-normalize-input-interfaces}, \\
\text{compress-normalize-output-interfaces}] \Longrightarrow \\
\text{normalized-nnf-match } m \Longrightarrow f m = \text{Some } m' \Longrightarrow \text{normalized-nnf-match } m'$$

<proof> **lemma** *compress-normalize-besteffort-matches:*

assumes *generic: primitive-matcher-generic* β

shows $f \in \text{set} [\text{compress-normalize-protocols}, \\
\text{compress-normalize-input-interfaces}, \\
\text{compress-normalize-output-interfaces}] \Longrightarrow \\
\text{normalized-nnf-match } m \Longrightarrow$

$$f m = \text{Some } m' \Longrightarrow \\
\text{matches } (\beta, \alpha) m' a p = \text{matches } (\beta, \alpha) m a p$$

<proof>

lemma *compress-normalize-besteffort-Some:*

assumes *generic: primitive-matcher-generic* β

shows $\text{normalized-nnf-match } m \Longrightarrow$

$\text{compress-normalize-besteffort } m = \text{Some } m' \implies$
 $\text{matches } (\beta, \alpha) m' a p = \text{matches } (\beta, \alpha) m a p$
 ⟨proof⟩

lemma *compress-normalize-besteffort-None:*
assumes *generic: primitive-matcher-generic* β
shows $\text{normalized-nnf-match } m \implies$
 $\text{compress-normalize-besteffort } m = \text{None} \implies$
 $\neg \text{matches } (\beta, \alpha) m a p$
 ⟨proof⟩

lemma *compress-normalize-besteffort-nnf:*
 $\text{normalized-nnf-match } m \implies$
 $\text{compress-normalize-besteffort } m = \text{Some } m' \implies$
 $\text{normalized-nnf-match } m'$
 ⟨proof⟩

lemma *compress-normalize-besteffort-not-introduces-Iiface:*
 $\neg \text{has-disc is-Iiface } m \implies \text{normalized-nnf-match } m \implies \text{compress-normalize-besteffort}$
 $m = \text{Some } m' \implies$
 $\neg \text{has-disc is-Iiface } m'$
 ⟨proof⟩

lemma *compress-normalize-besteffort-not-introduces-Oiface:*
 $\neg \text{has-disc is-Oiface } m \implies \text{normalized-nnf-match } m \implies \text{compress-normalize-besteffort}$
 $m = \text{Some } m' \implies$
 $\neg \text{has-disc is-Oiface } m'$
 ⟨proof⟩

lemma *compress-normalize-besteffort-not-introduces-Iiface-negated:*
 $\neg \text{has-disc-negated is-Iiface False } m \implies \text{normalized-nnf-match } m \implies \text{com-}$
 $\text{press-normalize-besteffort } m = \text{Some } m' \implies$
 $\neg \text{has-disc-negated is-Iiface False } m'$
 ⟨proof⟩

lemma *compress-normalize-besteffort-not-introduces-Oiface-negated:*
 $\neg \text{has-disc-negated is-Oiface False } m \implies \text{normalized-nnf-match } m \implies \text{com-}$
 $\text{press-normalize-besteffort } m = \text{Some } m' \implies$
 $\neg \text{has-disc-negated is-Oiface False } m'$
 ⟨proof⟩

lemma *compress-normalize-besteffort-not-introduces-Prot-negated:*
 $\neg \text{has-disc-negated is-Prot False } m \implies \text{normalized-nnf-match } m \implies \text{com-}$
 $\text{press-normalize-besteffort } m = \text{Some } m' \implies$
 $\neg \text{has-disc-negated is-Prot False } m'$
 ⟨proof⟩

lemma *compress-normalize-besteffort-hasdisc:*
 $\neg \text{has-disc disc } m \implies (\forall a. \neg \text{disc } (\text{Iiface } a)) \implies (\forall a. \neg \text{disc } (\text{Oiface } a)) \implies$
 $(\forall a. \neg \text{disc } (\text{Prot } a)) \implies$
 $\text{normalized-nnf-match } m \implies \text{compress-normalize-besteffort } m = \text{Some } m'$
 \implies
 $\text{normalized-nnf-match } m' \wedge \neg \text{has-disc disc } m'$
 ⟨proof⟩

lemma *compress-normalize-besteffort-hasdisc-negated:*

$\neg \text{has-disc-negated disc False } m \implies$
 $(\forall a. \neg \text{disc (Iface } a)) \implies (\forall a. \neg \text{disc (Oiface } a)) \implies (\forall a. \neg \text{disc (Prot } a)) \implies$
 $\text{normalized-nnf-match } m \implies \text{compress-normalize-besteffort } m = \text{Some } m'$
 \implies
 $\text{normalized-nnf-match } m' \wedge \neg \text{has-disc-negated disc False } m'$

<proof>

lemma *compress-normalize-besteffort-preserves-normalized-n-primitive:*

$\text{normalized-n-primitive (disc, sel) } P m \implies$
 $(\forall a. \neg \text{disc (Iface } a)) \implies (\forall a. \neg \text{disc (Oiface } a)) \implies (\forall a. \neg \text{disc (Prot } a))$
 \implies
 $\text{normalized-nnf-match } m \implies \text{compress-normalize-besteffort } m = \text{Some } m' \implies$
 $\text{normalized-nnf-match } m' \wedge \text{normalized-n-primitive (disc, sel) } P m'$
<proof>

end

39 Transforming rulesets

39.1 Optimizations

lemma *approximating-bigstep-fun-remdups-rev:*

$\text{approximating-bigstep-fun } \gamma p (\text{remdups-rev } rs) s = \text{approximating-bigstep-fun } \gamma$
 $p rs s$
<proof>

lemma *remdups-rev-simplers: simple-ruleset rs \implies simple-ruleset (remdups-rev rs)*

<proof>

lemma *remdups-rev-preserve-matches:*

$\forall r \in \text{set } rs. P (\text{get-match } r) \implies \forall r \in \text{set } (\text{remdups-rev } rs). P (\text{get-match } r)$
<proof>

39.2 Optimize and Normalize to NNF form

definition *transform-optimize-dnf-strict* :: *'i::len common-primitive rule list \Rightarrow 'i common-primitive rule list* **where**

$\text{transform-optimize-dnf-strict} = \text{cut-off-after-match-any} \circ$
 $(\text{optimize-matches } \text{opt-MatchAny-match-expr} \circ$
 $\text{normalize-rules-dnf} \circ (\text{optimize-matches } (\text{opt-MatchAny-match-expr} \circ \text{optimize-primitive-univ})))$

theorem *transform-optimize-dnf-strict-structure:*

assumes *simplers: simple-ruleset rs* **and** *wf α : wf-unknown-match-tac α*

shows *simple-ruleset (transform-optimize-dnf-strict rs)*

and $\forall r \in \text{set } rs. \neg \text{has-disc disc (get-match } r) \implies$

$\forall r \in \text{set } (\text{transform-optimize-dnf-strict } rs). \neg \text{has-disc disc (get-match } r)$

and $\forall r \in \text{set } (\text{transform-optimize-dnf-strict } rs). \text{normalized-nnf-match } (\text{get-match } r)$
and $\forall r \in \text{set } rs. \text{normalized-n-primitive disc-sel } f (\text{get-match } r) \implies$
 $\forall r \in \text{set } (\text{transform-optimize-dnf-strict } rs). \text{normalized-n-primitive disc-sel } f (\text{get-match } r)$
and $\forall r \in \text{set } rs. \neg \text{has-disc-negated disc neg } (\text{get-match } r) \implies$
 $\forall r \in \text{set } (\text{transform-optimize-dnf-strict } rs). \neg \text{has-disc-negated disc neg } (\text{get-match } r)$
 $\langle \text{proof} \rangle$

theorem *transform-optimize-dnf-strict*:

assumes *simplers: simple-ruleset* *rs* **and** *wf* α : *wf-unknown-match-tac* α
shows $(\text{common-matcher}, \alpha), p \vdash \langle \text{transform-optimize-dnf-strict } rs, s \rangle \Rightarrow_{\alpha} t \longleftrightarrow$
 $(\text{common-matcher}, \alpha), p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t$
 $\langle \text{proof} \rangle$

39.3 Abstracting over unknowns

definition *transform-remove-unknowns-generic*

$:: ('a, 'packet) \text{match-tac} \Rightarrow 'a \text{ rule list} \Rightarrow 'a \text{ rule list}$

where

$\text{transform-remove-unknowns-generic } \gamma = \text{optimize-matches-a } (\text{remove-unknowns-generic } \gamma)$

theorem *transform-remove-unknowns-generic*:

assumes *simplers: simple-ruleset* *rs*
and *wf* α : *wf-unknown-match-tac* α **and** *packet-independent- α* : *packet-independent- α*
 α
and *wf* β : *packet-independent- β -unknown* β
shows $(\beta, \alpha), p \vdash \langle \text{transform-remove-unknowns-generic } (\beta, \alpha) rs, s \rangle \Rightarrow_{\alpha} t \longleftrightarrow$
 $(\beta, \alpha), p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t$
and *simple-ruleset* $(\text{transform-remove-unknowns-generic } (\beta, \alpha) rs)$
and $\forall r \in \text{set } rs. \neg \text{has-disc disc } (\text{get-match } r) \implies$
 $\forall r \in \text{set } (\text{transform-remove-unknowns-generic } (\beta, \alpha) rs). \neg \text{has-disc disc } (\text{get-match } r)$
and $\forall r \in \text{set } (\text{transform-remove-unknowns-generic } (\beta, \alpha) rs). \neg \text{has-unknowns } \beta (\text{get-match } r)$

and $\forall r \in \text{set } rs. \text{normalized-n-primitive disc-sel } f (\text{get-match } r) \implies$
 $\forall r \in \text{set } (\text{transform-remove-unknowns-generic } (\beta, \alpha) rs). \text{normalized-n-primitive disc-sel } f (\text{get-match } r)$
and $\forall r \in \text{set } rs. \neg \text{has-disc-negated disc neg } (\text{get-match } r) \implies$
 $\forall r \in \text{set } (\text{transform-remove-unknowns-generic } (\beta, \alpha) rs). \neg \text{has-disc-negated disc neg } (\text{get-match } r)$
 $\langle \text{proof} \rangle$

thm *transform-remove-unknowns-generic*[*OF* - - - *packet-independent- β -unknown-common-matcher*]

corollary *transform-remove-unknowns-upper*: **defines** *upper* $\equiv \text{optimize-matches-a}$

upper-closure-matchexpr

assumes *simplers: simple-ruleset rs*

shows $(\text{common-matcher}, \text{in-doubt-allow}), p \vdash \langle \text{upper } rs, s \rangle \Rightarrow_{\alpha} t \longleftrightarrow (\text{common-matcher}, \text{in-doubt-allow}), p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t$

and *simple-ruleset (upper rs)*

and $\forall r \in \text{set } rs. \neg \text{has-disc disc (get-match } r) \implies$

$\forall r \in \text{set (upper } rs). \neg \text{has-disc disc (get-match } r)$

and $\forall r \in \text{set (upper } rs). \neg \text{has-disc is-Extra (get-match } r)$

and $\forall r \in \text{set } rs. \text{normalized-n-primitive disc-sel } f \text{ (get-match } r) \implies$

$\forall r \in \text{set (upper } rs). \text{normalized-n-primitive disc-sel } f \text{ (get-match } r)$

and $\forall r \in \text{set } rs. \neg \text{has-disc-negated disc neg (get-match } r) \implies$

$\forall r \in \text{set (upper } rs). \neg \text{has-disc-negated disc neg (get-match } r)$

<proof>

corollary *transform-remove-unknowns-lower: defines lower \equiv optimize-matches-a*

lower-closure-matchexpr

assumes *simplers: simple-ruleset rs*

shows $(\text{common-matcher}, \text{in-doubt-deny}), p \vdash \langle \text{lower } rs, s \rangle \Rightarrow_{\alpha} t \longleftrightarrow (\text{common-matcher}, \text{in-doubt-deny}), p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t$

and *simple-ruleset (lower rs)*

and $\forall r \in \text{set } rs. \neg \text{has-disc disc (get-match } r) \implies$

$\forall r \in \text{set (lower } rs). \neg \text{has-disc disc (get-match } r)$

and $\forall r \in \text{set (lower } rs). \neg \text{has-disc is-Extra (get-match } r)$

and $\forall r \in \text{set } rs. \text{normalized-n-primitive disc-sel } f \text{ (get-match } r) \implies$

$\forall r \in \text{set (lower } rs). \text{normalized-n-primitive disc-sel } f \text{ (get-match } r)$

and $\forall r \in \text{set } rs. \neg \text{has-disc-negated disc neg (get-match } r) \implies$

$\forall r \in \text{set (lower } rs). \neg \text{has-disc-negated disc neg (get-match } r)$

<proof>

39.4 Normalizing and Transforming Primitives

Rewrite the primitives IPs and Ports such that can be used by the simple firewall.

definition *transform-normalize-primitives :: 'i::len common-primitive rule list \Rightarrow 'i common-primitive rule list* **where**

transform-normalize-primitives =

optimize-matches-option compress-normalize-besteffort \circ — normalizes protocols, needs to go last

normalize-rules normalize-dst-ips \circ

normalize-rules normalize-src-ips \circ

normalize-rules normalize-dst-ports \circ — may introduce new matches on protocols

normalize-rules normalize-src-ports \circ — may introduce new matches in protocols

normalize-rules rewrite-MultiportPorts — introduces *Src-Ports* and *Dst-Ports* matches

thm *normalize-primitive-extract-preserves-unrelated-normalized-n-primitive*
lemma *normalize-rules-preserves-unrelated-normalized-n-primitive:*
assumes $\forall r \in \text{set } rs. \text{normalized-nnf-match } (\text{get-match } r) \wedge \text{normalized-n-primitive}$
 $(\text{disc2}, \text{sel2}) P (\text{get-match } r)$
and $\text{wf-disc-sel } (\text{disc1}, \text{sel1}) C$
and $\forall a. \neg \text{disc2 } (C a)$
shows $\forall r \in \text{set } (\text{normalize-rules } (\text{normalize-primitive-extract } (\text{disc1}, \text{sel1}) C$
 $f) rs).$
 $\text{normalized-nnf-match } (\text{get-match } r) \wedge \text{normalized-n-primitive } (\text{disc2},$
 $\text{sel2}) P (\text{get-match } r)$
thm *normalize-rules-preserves* [**where** $P = \lambda m. \text{normalized-nnf-match } m \wedge \text{normalized-n-primitive } (\text{disc2}, \text{sel2}) P m$
and $f = \text{normalize-primitive-extract } (\text{disc1}, \text{sel1}) C f]$
 $\langle \text{proof} \rangle$

lemma *normalize-rules-normalized-n-primitive:*
assumes $\forall r \in \text{set } rs. \text{normalized-nnf-match } (\text{get-match } r)$
and $\forall m. \text{normalized-nnf-match } m \longrightarrow$
 $(\forall m' \in \text{set } (\text{normalize-primitive-extract } (\text{disc}, \text{sel}) C f m). \text{normalized-n-primitive } (\text{disc}, \text{sel}) P m')$
shows $\forall r \in \text{set } (\text{normalize-rules } (\text{normalize-primitive-extract } (\text{disc}, \text{sel}) C f)$
 $rs).$
 $\text{normalized-n-primitive } (\text{disc}, \text{sel}) P (\text{get-match } r)$
 $\langle \text{proof} \rangle$

lemma *optimize-matches-option-compress-normalize-besteffort-preserves-unrelated-normalized-n-primitive:*
assumes $\forall r \in \text{set } rs. \text{normalized-nnf-match } (\text{get-match } r) \wedge \text{normalized-n-primitive}$
 $(\text{disc2}, \text{sel2}) P (\text{get-match } r)$
and $\forall a. \neg \text{disc2 } (\text{Iiface } a)$ **and** $\forall a. \neg \text{disc2 } (\text{Oiface } a)$ **and** $\forall a. \neg \text{disc2}$
 $(\text{Prot } a)$
shows $\forall r \in \text{set } (\text{optimize-matches-option } \text{compress-normalize-besteffort } rs).$
 $\text{normalized-nnf-match } (\text{get-match } r) \wedge \text{normalized-n-primitive } (\text{disc2},$
 $\text{sel2}) P (\text{get-match } r)$
thm *optimize-matches-option-preserves*
 $\langle \text{proof} \rangle$

theorem *transform-normalize-primitives:*
— all discriminators which will not be normalized remain unchanged
defines $\text{unchanged disc} \equiv (\forall a. \neg \text{disc } (\text{Src-Ports } a)) \wedge (\forall a. \neg \text{disc } (\text{Dst-Ports}$
 $a)) \wedge$
 $(\forall a. \neg \text{disc } (\text{Src } a)) \wedge (\forall a. \neg \text{disc } (\text{Dst } a))$
— also holds for these discriminators, but not for *Prot*, which might be changed
and $\text{changeddisc} \equiv ((\forall a. \neg \text{disc } (\text{Iiface } a)) \vee \text{disc} = \text{is-Iiface}) \wedge$
 $(\forall a. \neg \text{disc } (\text{Oiface } a)) \vee \text{disc} = \text{is-Oiface})$

assumes *simplers: simple-ruleset* $(rs :: 'i::\text{len common-primitive rule list})$

and *wf* α : *wf-unknown-match-tac* α
and *normalized*: $\forall r \in \text{set } rs. \text{normalized-nnf-match } (\text{get-match } r)$
shows $(\text{common-matcher}, \alpha), p \vdash \langle \text{transform-normalize-primitives } rs, s \rangle \Rightarrow_{\alpha} t \iff$
 $(\text{common-matcher}, \alpha), p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t$
and *simple-ruleset* $(\text{transform-normalize-primitives } rs)$
and *unchanged disc1* $\implies \text{changeddisc } disc1 \implies \forall a. \neg disc1 (\text{Prot } a) \implies$
 $\forall r \in \text{set } rs. \neg \text{has-disc } disc1 (\text{get-match } r) \implies$
 $\forall r \in \text{set } (\text{transform-normalize-primitives } rs). \neg \text{has-disc } disc1 (\text{get-match } r)$
and $\forall r \in \text{set } (\text{transform-normalize-primitives } rs). \text{normalized-nnf-match } (\text{get-match } r)$
and $\forall r \in \text{set } (\text{transform-normalize-primitives } rs).$
 $\text{normalized-src-ports } (\text{get-match } r) \wedge \text{normalized-dst-ports } (\text{get-match } r) \wedge$
 $\text{normalized-src-ips } (\text{get-match } r) \wedge \text{normalized-dst-ips } (\text{get-match } r) \wedge$
 $\neg \text{has-disc is-MultiportPorts } (\text{get-match } r)$
and *unchanged disc2* $\implies (\forall a. \neg disc2 (\text{Iface } a)) \implies (\forall a. \neg disc2 (\text{Oiface } a)) \implies$
 $(\forall a. \neg disc2 (\text{Prot } a)) \implies$
 $\forall r \in \text{set } rs. \text{normalized-n-primitive } (disc2, sel2) f (\text{get-match } r) \implies$
 $\forall r \in \text{set } (\text{transform-normalize-primitives } rs). \text{normalized-n-primitive } (disc2, sel2) f (\text{get-match } r)$

— For *disc3*, we do not allow ports and ips, because these are changed. Here is the complicated part: (It is only complicated if, basically *disc3* is *is-Prot*) In addition, either it must not be protocol or (complicated case) there must be no negated port matches in the ruleset. Note that negated *Src-Ports* or *Dst-Ports* can also be introduced by rewriting *MultiportPorts*

and *unchanged disc3* $\implies \text{changeddisc } disc3 \implies$
 $(\forall a. \neg disc3 (\text{Prot } a)) \vee$
 $(disc3 = \text{is-Prot} \wedge (\forall r \in \text{set } rs.$
 $\neg \text{has-disc-negated is-Src-Ports False } (\text{get-match } r) \wedge$
 $\neg \text{has-disc-negated is-Dst-Ports False } (\text{get-match } r) \wedge$
 $\neg \text{has-disc is-MultiportPorts } (\text{get-match } r))) \implies$
 $\forall r \in \text{set } rs. \neg \text{has-disc-negated } disc3 \text{ False } (\text{get-match } r) \implies$
 $\forall r \in \text{set } (\text{transform-normalize-primitives } rs). \neg \text{has-disc-negated } disc3$
 $\text{False } (\text{get-match } r)$
 $\langle \text{proof} \rangle$

theorem *iiface-constrain*:

assumes *simplers*: *simple-ruleset* rs
and *normalized*: $\forall r \in \text{set } rs. \text{normalized-nnf-match } (\text{get-match } r)$
and *wf-ipassmt*: *ipassmt-sanity-nowildcards* $ipassmt$
and *nospoofing*: $\bigwedge ips. ipassmt (\text{Iface } (p\text{-iiface } p)) = \text{Some } ips \implies p\text{-src } p \in$
 $ipcidr\text{-union-set } (\text{set } ips)$
shows $(\text{common-matcher}, \alpha), p \vdash \langle \text{optimize-matches } (\text{iiface-constrain } ipassmt) rs,$
 $s \rangle \Rightarrow_{\alpha} t \iff (\text{common-matcher}, \alpha), p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t$
and *simple-ruleset* $(\text{optimize-matches } (\text{iiface-constrain } ipassmt) rs)$
 $\langle \text{proof} \rangle$

In contrast to $\llbracket \text{simple-ruleset } ?rs; \forall r \in \text{set } ?rs. \text{normalized-nnf-match } (\text{get-match } r) \rrbracket$

r); $ipassmt$ -sanity-nowildcards $?ipassmt$; $\wedge ips. ?ipassmt$ ($Iface$ (p -iface $?p$))
 $= Some\ ips \implies p$ -src $?p \in ipcidr$ -union-set ($set\ ips$) \implies ($common$ -matcher,
 $?a$), $?p \vdash \langle optimize$ -matches ($iface$ -constrain $?ipassmt$) $?rs, ?s \rangle \Rightarrow_\alpha ?t =$
 $(common$ -matcher, $?a$), $?p \vdash \langle ?rs, ?s \rangle \Rightarrow_\alpha ?t$

$\llbracket simple$ -ruleset $?rs; \forall r \in set\ ?rs. normalized$ -nnf-match (get -match r); $ipassmt$ -sanity-nowildcards
 $?ipassmt; \wedge ips. ?ipassmt$ ($Iface$ (p -iface $?p$)) $= Some\ ips \implies p$ -src $?p \in ip$ -
 $cidr$ -union-set ($set\ ips$) $\implies simple$ -ruleset ($optimize$ -matches ($iface$ -constrain
 $?ipassmt$) $?rs$), this requires $ipassmt$ -sanity-disjoint and as much stronger
nospoof assumption: This assumption requires that the packet is actually in
 $ipassmt$!

theorem *iface-rewrite*:

assumes *simplers*: $simple$ -ruleset rs
and *normalized*: $\forall r \in set\ rs. normalized$ -nnf-match (get -match r)
and *wf-ipassmt*: $ipassmt$ -sanity-nowildcards $ipassmt$
and *disjoint-ipassmt*: $ipassmt$ -sanity-disjoint $ipassmt$
and *nospoofing*: $\exists ips. ipassmt$ ($Iface$ (p -iface p)) $= Some\ ips \wedge p$ -src $p \in$
 $ipcidr$ -union-set ($set\ ips$)
shows ($common$ -matcher, α), $p \vdash \langle optimize$ -matches ($iface$ -rewrite $ipassmt$) $rs,$
 $s \rangle \Rightarrow_\alpha t \iff (common$ -matcher, α), $p \vdash \langle rs, s \rangle \Rightarrow_\alpha t$
and $simple$ -ruleset ($optimize$ -matches ($iface$ -rewrite $ipassmt$) rs)
 $\langle proof \rangle$

theorem *oiface-rewrite*:

assumes *simplers*: $simple$ -ruleset rs
and *normalized*: $\forall r \in set\ rs. normalized$ -nnf-match (get -match r)
and *wf-ipassmt*: $ipassmt$ -sanity-nowildcards $ipassmt$
and *ipassmt-from-rt*: $ipassmt = map$ -of ($routing$ - $ipassmt\ rt$)
and *correct-routing*: $correct$ -routing rt
and *rtbl-decided*: $output$ -iface ($routing$ -table- $semantics\ rt$ (p -dst p)) $= p$ -oiface
 p
shows ($common$ -matcher, α), $p \vdash \langle optimize$ -matches ($oiface$ -rewrite $ipassmt$) $rs,$
 $s \rangle \Rightarrow_\alpha t \iff (common$ -matcher, α), $p \vdash \langle rs, s \rangle \Rightarrow_\alpha t$
and $simple$ -ruleset ($optimize$ -matches ($oiface$ -rewrite $ipassmt$) rs)
 $\langle proof \rangle$

definition *upper-closure* :: $'i::len$ $common$ -primitive rule list $\Rightarrow 'i$ $common$ -primitive
rule list **where**

$upper$ -closure $rs == remdups$ -rev ($transform$ -optimize-dnf-strict
($transform$ -normalize-primitives ($transform$ -optimize-dnf-strict ($optimize$ -matches-a
 $upper$ -closure-matchexpr rs))))

definition *lower-closure* :: $'i::len$ $common$ -primitive rule list $\Rightarrow 'i$ $common$ -primitive
rule list **where**

$lower$ -closure $rs == remdups$ -rev ($transform$ -optimize-dnf-strict
($transform$ -normalize-primitives ($transform$ -optimize-dnf-strict ($optimize$ -matches-a
 $lower$ -closure-matchexpr rs))))

putting it all together

lemma *transform-upper-closure*:

assumes *simplers*: *simple-ruleset* *rs*

— semantics are preserved

shows $(\text{common-matcher}, \text{in-doubt-allow}), p \vdash \langle \text{upper-closure } rs, s \rangle \Rightarrow_{\alpha} t \iff$
 $(\text{common-matcher}, \text{in-doubt-allow}), p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t$

and *simple-ruleset* (*upper-closure* *rs*)

— simple, normalized rules without unknowns

and $\forall r \in \text{set } (\text{upper-closure } rs). \text{normalized-nnf-match } (\text{get-match } r) \wedge$
 $\text{normalized-src-ports } (\text{get-match } r) \wedge$
 $\text{normalized-dst-ports } (\text{get-match } r) \wedge$
 $\text{normalized-src-ips } (\text{get-match } r) \wedge$
 $\text{normalized-dst-ips } (\text{get-match } r) \wedge$
 $\neg \text{has-disc is-MultiportPorts } (\text{get-match } r) \wedge$
 $\neg \text{has-disc is-Extra } (\text{get-match } r)$

— no new primitives are introduced

and $\forall a. \neg \text{disc } (\text{Src-Ports } a) \implies \forall a. \neg \text{disc } (\text{Dst-Ports } a) \implies \forall a. \neg \text{disc } (\text{Src } a) \implies \forall a. \neg \text{disc } (\text{Dst } a) \implies$

$\forall a. \neg \text{disc } (\text{Iiface } a) \vee \text{disc} = \text{is-Iiface} \implies \forall a. \neg \text{disc } (\text{Oiface } a) \vee \text{disc} = \text{is-Oiface} \implies$

$\forall a. \neg \text{disc } (\text{Prot } a) \implies$

$\forall r \in \text{set } rs. \neg \text{has-disc disc } (\text{get-match } r) \implies \forall r \in \text{set } (\text{upper-closure } rs). \neg \text{has-disc disc } (\text{get-match } r)$

and $\forall a. \neg \text{disc } (\text{Src-Ports } a) \implies \forall a. \neg \text{disc } (\text{Dst-Ports } a) \implies \forall a. \neg \text{disc } (\text{Src } a) \implies \forall a. \neg \text{disc } (\text{Dst } a) \implies$

$\forall a. \neg \text{disc } (\text{Iiface } a) \vee \text{disc} = \text{is-Iiface} \implies \forall a. \neg \text{disc } (\text{Oiface } a) \vee \text{disc} = \text{is-Oiface} \implies$

$(\forall a. \neg \text{disc } (\text{Prot } a)) \vee$

$\text{disc} = \text{is-Prot} \wedge$ — if it is prot, there must not be negated matches on ports

$(\forall r \in \text{set } rs. \neg \text{has-disc-negated is-Src-Ports False } (\text{get-match } r) \wedge$

$\neg \text{has-disc-negated is-Dst-Ports False } (\text{get-match } r) \wedge$

$\neg \text{has-disc is-MultiportPorts } (\text{get-match } r)) \implies$

$\forall r \in \text{set } rs. \neg \text{has-disc-negated disc False } (\text{get-match } r) \implies$

$\forall r \in \text{set } (\text{upper-closure } rs). \neg \text{has-disc-negated disc False } (\text{get-match } r)$

<proof>

lemma *transform-lower-closure*:

assumes *simplers*: *simple-ruleset* *rs*

— semantics are preserved

shows $(\text{common-matcher}, \text{in-doubt-deny}), p \vdash \langle \text{lower-closure } rs, s \rangle \Rightarrow_{\alpha} t \iff$
 $(\text{common-matcher}, \text{in-doubt-deny}), p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t$

and *simple-ruleset* (*lower-closure* *rs*)

— simple, normalized rules without unknowns

and $\forall r \in \text{set } (\text{lower-closure } rs). \text{normalized-nnf-match } (\text{get-match } r) \wedge$
 $\text{normalized-src-ports } (\text{get-match } r) \wedge$
 $\text{normalized-dst-ports } (\text{get-match } r) \wedge$
 $\text{normalized-src-ips } (\text{get-match } r) \wedge$

$$\begin{aligned}
& \text{normalized-dst-ips (get-match r)} \wedge \\
& \neg \text{has-disc is-MultiportPorts (get-match r)} \wedge \\
& \neg \text{has-disc is-Extra (get-match r)} \\
- & \text{ no new primitives are introduced} \\
\mathbf{and} & \forall a. \neg \text{disc (Src-Ports a)} \implies \forall a. \neg \text{disc (Dst-Ports a)} \implies \forall a. \neg \text{disc (Src} \\
a) & \implies \forall a. \neg \text{disc (Dst a)} \implies \\
& \forall a. \neg \text{disc (Iiface a)} \vee \text{disc} = \text{is-Iiface} \implies \forall a. \neg \text{disc (Oiface a)} \vee \text{disc} = \\
\text{is-Oiface} & \implies \\
& \forall a. \neg \text{disc (Prot a)} \implies \\
& \forall r \in \text{set rs}. \neg \text{has-disc disc (get-match r)} \implies \\
& \forall r \in \text{set (lower-closure rs)}. \neg \text{has-disc disc (get-match r)} \\
\mathbf{and} & \forall a. \neg \text{disc (Src-Ports a)} \implies \forall a. \neg \text{disc (Dst-Ports a)} \implies \forall a. \neg \text{disc (Src} \\
a) & \implies \forall a. \neg \text{disc (Dst a)} \implies \\
& \forall a. \neg \text{disc (Iiface a)} \vee \text{disc} = \text{is-Iiface} \implies \forall a. \neg \text{disc (Oiface a)} \vee \text{disc} = \\
\text{is-Oiface} & \implies \\
& (\forall a. \neg \text{disc (Prot a)}) \vee \text{disc} = \text{is-Prot} \wedge \\
& (\forall r \in \text{set rs}. \neg \text{has-disc-negated is-Src-Ports False (get-match r)} \wedge \\
& \quad \neg \text{has-disc-negated is-Dst-Ports False (get-match r)} \wedge \\
& \quad \neg \text{has-disc is-MultiportPorts (get-match r)}) \implies \\
& \forall r \in \text{set rs}. \neg \text{has-disc-negated disc False (get-match r)} \implies \\
& \forall r \in \text{set (lower-closure rs)}. \neg \text{has-disc-negated disc False (get-match r)} \\
\langle \text{proof} \rangle &
\end{aligned}$$

definition *iface-try-rewrite*

$$\begin{aligned}
& :: (\text{iface} \times ('i::\text{len word} \times \text{nat}) \text{list}) \text{list} \\
& \Rightarrow 'i \text{ prefix-routing option} \\
& \Rightarrow 'i \text{ common-primitive rule list} \\
& \Rightarrow 'i \text{ common-primitive rule list}
\end{aligned}$$

where

$$\begin{aligned}
& \text{iface-try-rewrite ipassmt rtblo rs} \equiv \\
& \text{let } o\text{-rewrite} = (\text{case rtblo of None} \Rightarrow \text{id} \mid \text{Some rtbl} \Rightarrow \\
& \quad \text{transform-optimize-dnf-strict} \circ \text{optimize-matches (oiface-rewrite (map-of-ipassmt} \\
& \quad \text{(routing-ipassmt rtbl))))}) \text{ in} \\
& \text{if ipassmt-sanity-disjoint (map-of ipassmt)} \wedge \text{ipassmt-sanity-defined rs (map-of} \\
& \text{ipassmt)} \text{ then} \\
& \quad \text{optimize-matches (iiface-rewrite (map-of-ipassmt ipassmt)) (o-rewrite rs)} \\
& \text{else} \\
& \quad \text{optimize-matches (iiface-constrain (map-of-ipassmt ipassmt)) (o-rewrite rs)}
\end{aligned}$$

Where $(\text{iface} \times ('i \text{ word} \times \text{nat}) \text{list}) \text{list}$ is *map-of 'i ipassignment*. The sanity checkers need to iterate over the interfaces, hence we don't pass a map but a list of tuples.

In `Transform.thy` there should be the final correctness theorem for *iface-try-rewrite*. Here are some structural properties.

lemma *iface-try-rewrite-simplers: simple-ruleset rs \implies simple-ruleset (iface-try-rewrite ipassmt rtblo rs)*

$\langle \text{proof} \rangle$

lemma *iiface-rewrite-preserves-nodisc:*

$\forall a. \neg \text{disc} (\text{Src } a) \implies \neg \text{has-disc disc } m \implies \neg \text{has-disc disc} (\text{iiface-rewrite ipassmt } m)$
<proof>

lemma *iiface-constrain-preserves-nodisc:*

$\forall a. \neg \text{disc} (\text{Src } a) \implies \neg \text{has-disc disc } m \implies \neg \text{has-disc disc} (\text{iiface-constrain ipassmt } m)$
<proof>

lemma *iface-try-rewrite-preserves-nodisc:*

simple-ruleset rs \implies
 $\forall a. \neg \text{disc} (\text{Src } a) \implies \forall a. \neg \text{disc} (\text{Dst } a) \implies$
 $\forall r \in \text{set } rs. \neg \text{has-disc disc} (\text{get-match } r) \implies$
 $\forall r \in \text{set} (\text{iface-try-rewrite ipassmt rtblo } rs). \neg \text{has-disc disc} (\text{get-match } r)$
<proof>

theorem *iface-try-rewrite-no-rtbl:*

assumes *simplers: simple-ruleset rs*
and *normalized:* $\forall r \in \text{set } rs. \text{normalized-nnf-match} (\text{get-match } r)$
and *wf-ipassmt1:* *ipassmt-sanity-nowildcards* (*map-of ipassmt*) **and** *wf-ipassmt2:* *distinct* (*map fst ipassmt*)
and *nospoofing:* $\exists \text{ips}. (\text{map-of ipassmt}) (\text{Iface } (p\text{-iiface } p)) = \text{Some ips} \wedge p\text{-src } p \in \text{ipcidr-union-set} (\text{set ips})$
shows (*common-matcher*, α), $p \vdash \langle \text{iface-try-rewrite ipassmt None } rs, s \rangle \Rightarrow_{\alpha} t \longleftrightarrow$
(*common-matcher*, α), $p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t$
<proof>

lemma *optimize-matches-comp:*

assumes *mono:* $\bigwedge m. \text{matcheq-matchNone } m \implies \text{matcheq-matchNone} (g \ m)$
shows *optimize-matches* ($g \circ f$) *rs* = *optimize-matches* g (*optimize-matches* f) *rs*)
<proof>

context *begin*

private lemma *iiface-rewrite-monoNone:* *matcheq-matchNone* $m \implies \text{matcheq-matchNone} (\text{iiface-rewrite ipassmt } m)$

<proof> **lemma** *iiface-constrain-monoNone:* *matcheq-matchNone* $m \implies \text{matcheq-matchNone} (\text{iiface-constrain ipassmt } m)$

<proof> **lemmas** *optimize-matches-iiface-comp* = *optimize-matches-comp*[*OF iiface-rewrite-monoNone*]

optimize-matches-comp[*OF iiface-constrain-monoNone*]

end

theorem *iface-try-rewrite-rtbl*:
assumes *simplers*: *simple-ruleset* *rs*
and *normalized*: $\forall r \in \text{set } rs. \text{normalized-nnf-match } (\text{get-match } r)$
and *wf-ipassmt*: *ipassmt-sanity-nowildcards* (*map-of ipassmt*) *distinct* (*map fst ipassmt*)
and *nospoofing*: $\exists ips. (\text{map-of ipassmt}) (\text{Iface } (p\text{-iface } p)) = \text{Some } ips \wedge p\text{-src } p \in \text{ipcidr-union-set } (\text{set } ips)$
and *routing-decided*: *output-iface* (*routing-table-semantic*s *rtbl* (*p-dst* *p*)) = *p-iface* *p*
and *correct-routing*: *correct-routing* *rtbl*
and *wf-ipassmt-o*: *ipassmt-sanity-nowildcards* (*map-of* (*routing-ipassmt* *rtbl*))
and *wf-match-tac*: *wf-unknown-match-tac* α
shows (*common-matcher*, α), $p \vdash \langle \text{iface-try-rewrite ipassmt } (\text{Some } rtbl) rs, s \rangle \Rightarrow_{\alpha} t \longleftrightarrow (\text{common-matcher}, \alpha), p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t$
 $\langle \text{proof} \rangle$

end
theory *Conntrack-State-Transform*
imports *Common-Primitive-Matcher*
../Semantics-Ternary/Semantics-Ternary
begin

The following function assumes that the packet is in a certain state.

fun *ctstate-assume-state* :: *ctstate* \Rightarrow *'i::len common-primitive match-expr* \Rightarrow *'i common-primitive match-expr* **where**
ctstate-assume-state *s* (*Match* (*CT-State* *x*)) = (*if* $s \in x$ *then MatchAny* *else MatchNot MatchAny*) |
ctstate-assume-state *s* (*Match* *m*) = *Match* *m* |
ctstate-assume-state *s* (*MatchNot* *m*) = *MatchNot* (*ctstate-assume-state* *s* *m*) |
ctstate-assume-state - *MatchAny* = *MatchAny* |
ctstate-assume-state *s* (*MatchAnd* *m1* *m2*) = *MatchAnd* (*ctstate-assume-state* *s* *m1*) (*ctstate-assume-state* *s* *m2*)

lemma *ctstate-assume-state*: *p-tag-ctstate* $p = s \implies$
matches (*common-matcher*, α) (*ctstate-assume-state* *s* *m*) *a* *p* \longleftrightarrow *matches* (*common-matcher*, α) *m* *a* *p*
 $\langle \text{proof} \rangle$

definition *ctstate-assume-new* :: *'i::len common-primitive rule list* \Rightarrow *'i common-primitive rule list* **where**
ctstate-assume-new \equiv *optimize-matches* (*ctstate-assume-state* *CT-New*)

lemma *ctstate-assume-new-simple-ruleset*: *simple-ruleset* *rs* \implies *simple-ruleset* (*ctstate-assume-new* *rs*)
 $\langle \text{proof} \rangle$

Usually, the interesting part of a firewall is only about the rules for setting

up connections. That means, we mostly only care about packets in state *CT-New*. Use the function *ctstate-assume-new* to remove all state matching and just care about the connection setup.

corollary *ctstate-assume-new*: $p\text{-tag-ctstate } p = CT\text{-New} \implies$

$\text{approximating-bigstep-fun } (common\text{-matcher}, \alpha) p (ctstate\text{-assume-new } rs) s =$
 $\text{approximating-bigstep-fun } (common\text{-matcher}, \alpha) p rs s$
 ⟨proof⟩

If we assume the CT State is *CT-New*, we can also assume that the TCP SYN flag (*ipt-tcp-syn*) is set.

fun *ipt-tcp-flags-assume-flag* :: $ipt\text{-tcp-flags} \Rightarrow 'i::len \text{ common-primitive match-expr}$
 $\Rightarrow 'i \text{ common-primitive match-expr}$ **where**

$ipt\text{-tcp-flags-assume-flag } flg (Match (L4\text{-Flags } x)) = (if \text{ ipt-tcp-flags-equal } x \text{ flg}$
 $\text{ then } MatchAny \text{ else } (case \text{ match-tcp-flags-conjunct-option } x \text{ flg of } None \Rightarrow Match\text{-}$
 $Not \text{ MatchAny } | \text{ Some } f3 \Rightarrow Match (L4\text{-Flags } f3))) |$

$ipt\text{-tcp-flags-assume-flag } flg (Match m) = Match m |$

$ipt\text{-tcp-flags-assume-flag } flg (MatchNot m) = MatchNot (ipt\text{-tcp-flags-assume-flag}$
 $flg m) |$

$ipt\text{-tcp-flags-assume-flag} - MatchAny = MatchAny |$

$ipt\text{-tcp-flags-assume-flag } flg (MatchAnd m1 m2) = MatchAnd (ipt\text{-tcp-flags-assume-flag}$
 $flg m1) (ipt\text{-tcp-flags-assume-flag } flg m2)$

lemma *ipt-tcp-flags-assume-flag*: **assumes** $match\text{-tcp-flags } flg (p\text{-tcp-flags } p)$

shows $matches (common\text{-matcher}, \alpha) (ipt\text{-tcp-flags-assume-flag } flg m) a p \longleftrightarrow$
 $matches (common\text{-matcher}, \alpha) m a p$
 ⟨proof⟩

definition *ipt-tcp-flags-assume-syn* :: $'i::len \text{ common-primitive rule list} \Rightarrow 'i \text{ com-}$
 $mon\text{-primitive rule list}$ **where**

$ipt\text{-tcp-flags-assume-syn} \equiv optimize\text{-matches } (ipt\text{-tcp-flags-assume-flag } ipt\text{-tcp-syn})$

lemma *ipt-tcp-flags-assume-syn-simple-ruleset*: $simple\text{-ruleset } rs \implies simple\text{-ruleset}$
 $(ipt\text{-tcp-flags-assume-syn } rs)$

⟨proof⟩

corollary *ipt-tcp-flags-assume-syn*: $match\text{-tcp-flags } ipt\text{-tcp-syn} (p\text{-tcp-flags } p) \implies$

$\text{approximating-bigstep-fun } (common\text{-matcher}, \alpha) p (ipt\text{-tcp-flags-assume-syn } rs)$
 $s = \text{approximating-bigstep-fun } (common\text{-matcher}, \alpha) p rs s$
 ⟨proof⟩

definition *packet-assume-new* :: $'i::len \text{ common-primitive rule list} \Rightarrow 'i \text{ common-primitive}$
 $rule list$ **where**

$packet\text{-assume-new} \equiv ctstate\text{-assume-new} \circ ipt\text{-tcp-flags-assume-syn}$

lemma *packet-assume-new-simple-ruleset*: *simple-ruleset rs* \implies *simple-ruleset (packet-assume-new rs)*

<proof>

corollary *packet-assume-new*: *match-tcp-flags ipt-tcp-syn (p-tcp-flags p)* \implies *p-tag-ctstate p = CT-New* \implies

approximating-bigstep-fun (common-matcher, α) p (packet-assume-new rs) s = approximating-bigstep-fun (common-matcher, α) p rs s

<proof>

end

theory *Primitive-Abstract*

imports

Common-Primitive-toString

Transform

Conntrack-State-Transform

begin

40 Abstracting over Primitives

Abstract over certain primitives. The first parameter is a function *'i common-primitive negation-type* \Rightarrow *bool* to select the primitives to be abstracted over. The *'i common-primitive* is wrapped in a *'i common-primitive negation-type* to let the function selectively abstract only over negated, non-negated, or both kinds of primitives. This functions requires a *normalized-nnf-match*.

fun *abstract-primitive*

:: ('i::len common-primitive negation-type \Rightarrow bool) \Rightarrow 'i common-primitive match-expr \Rightarrow 'i common-primitive match-expr

where

abstract-primitive - MatchAny = MatchAny |

abstract-primitive disc (Match a) =

(if disc (Pos a)

then

Match (Extra (common-primitive-toString ipaddr-generic-toString a))

else

(Match a) |

abstract-primitive disc (MatchNot (Match a)) =

(if disc (Neg a)

then

Match (Extra (! "@common-primitive-toString ipaddr-generic-toString a))

$$\begin{aligned} & \text{else} \\ & (\text{MatchNot } (\text{Match } a)) \mid \\ & \text{abstract-primitive disc } (\text{MatchNot } m) = \text{MatchNot } (\text{abstract-primitive disc } m) \mid \\ & \text{abstract-primitive disc } (\text{MatchAnd } m1\ m2) = \text{MatchAnd } (\text{abstract-primitive disc } \\ & m1) (\text{abstract-primitive disc } m2) \end{aligned}$$

For example, a simple firewall requires that no negated interfaces and protocols occur in the expression.

definition *abstract-for-simple-firewall* :: 'i::len common-primitive match-expr \Rightarrow 'i common-primitive match-expr

where *abstract-for-simple-firewall* \equiv *abstract-primitive* ($\lambda r.$ case *r*
of *Pos* *a* \Rightarrow *is-CT-State* *a* \vee *is-L4-Flags* *a*
| *Neg* *a* \Rightarrow *is-Iiface* *a* \vee *is-Oiface* *a* \vee *is-Prot* *a* \vee *is-CT-State* *a* \vee
is-L4-Flags *a*)

lemma *abstract-primitive-preserves-normalized*:

$$\begin{aligned} \text{normalized-src-ports } m & \Longrightarrow \text{normalized-src-ports } (\text{abstract-primitive disc } m) \\ \text{normalized-dst-ports } m & \Longrightarrow \text{normalized-dst-ports } (\text{abstract-primitive disc } m) \\ \text{normalized-src-ips } m & \Longrightarrow \text{normalized-src-ips } (\text{abstract-primitive disc } m) \\ \text{normalized-dst-ips } m & \Longrightarrow \text{normalized-dst-ips } (\text{abstract-primitive disc } m) \\ \text{normalized-nnf-match } m & \Longrightarrow \text{normalized-nnf-match } (\text{abstract-primitive disc } m) \end{aligned}$$

<proof>

lemma *abstract-primitive-preserves-nodisc*:

$$\neg \text{has-disc } \text{disc}'\ m \Longrightarrow (\forall \text{str. } \neg \text{disc}' (\text{Extra } \text{str})) \Longrightarrow \neg \text{has-disc } \text{disc}' (\text{abstract-primitive disc } m)$$

<proof>

lemma *abstract-primitive-preserves-nodisc-negated*:

$$\neg \text{has-disc-negated } \text{disc}'\ \text{neg } m \Longrightarrow (\forall \text{str. } \neg \text{disc}' (\text{Extra } \text{str})) \Longrightarrow \neg \text{has-disc-negated } \text{disc}'\ \text{neg } (\text{abstract-primitive disc } m)$$

<proof>

lemma *abstract-primitive-nodisc*:

$$\forall x. \text{disc}'\ x \longrightarrow \text{disc } (\text{Pos } x) \wedge \text{disc } (\text{Neg } x) \Longrightarrow (\forall \text{str. } \neg \text{disc}' (\text{Extra } \text{str})) \Longrightarrow \neg \text{has-disc } \text{disc}' (\text{abstract-primitive disc } m)$$

<proof>

lemma *abstract-primitive-preserves-not-has-disc-negated*:

$$\forall a. \neg \text{disc } (\text{Extra } a) \Longrightarrow \neg \text{has-disc-negated } \text{disc } \text{neg } m \Longrightarrow \neg \text{has-disc-negated } \text{disc } \text{neg } (\text{abstract-primitive sel-f } m)$$

<proof>

lemma *abstract-for-simple-firewall-preserves-nodisc-negated*:

$$\forall a. \neg \text{disc } (\text{Extra } a) \Longrightarrow \neg \text{has-disc-negated } \text{disc } \text{False } m \Longrightarrow \neg \text{has-disc-negated } \text{disc } \text{False } (\text{abstract-for-simple-firewall } m)$$

<proof>

The function *ctstate-assume-state* can be used to fix a state and hence remove all state matches from the ruleset. It is therefore advisable to create a

simple firewall for a fixed state, e.g. with *ctstate-assume-new* before calling to *abstract-for-simple-firewall*.

lemma *not-hasdisc-ctstate-assume-state*: \neg *has-disc is-CT-State* (*ctstate-assume-state s m*)
 ⟨*proof*⟩

lemma *abstract-for-simple-firewall-hasdisc*: **fixes** *m* :: '*i*::*len* *common-primitive match-expr*

shows \neg *has-disc is-CT-State* (*abstract-for-simple-firewall m*)
and \neg *has-disc is-L4-Flags* (*abstract-for-simple-firewall m*)
 ⟨*proof*⟩

lemma *abstract-for-simple-firewall-negated-ifaces-prot*: **fixes** *m* :: '*i*::*len* *common-primitive match-expr*

shows *normalized-nnf-match m* \implies \neg *has-disc-negated* ($\lambda a. \text{is-Iiface } a \vee \text{is-Oiface } a$) *False* (*abstract-for-simple-firewall m*)
and *normalized-nnf-match m* \implies \neg *has-disc-negated is-Prot False* (*abstract-for-simple-firewall m*)
 ⟨*proof*⟩

context

begin

private lemma *abstract-primitive-in-doubt-allow-Allow*:

primitive-matcher-generic $\beta \implies$ *normalized-nnf-match m* \implies
matches (β , *in-doubt-allow*) *m action.Accept p* \implies
matches (β , *in-doubt-allow*) (*abstract-primitive disc m*) *action.Accept p*
 ⟨*proof*⟩ **lemma** *abstract-primitive-in-doubt-allow-Allow2*:

primitive-matcher-generic $\beta \implies$ *normalized-nnf-match m* \implies
 \neg *matches* (β , *in-doubt-allow*) *m action.Drop p* \implies
 \neg *matches* (β , *in-doubt-allow*) (*abstract-primitive disc m*) *action.Drop p*
 ⟨*proof*⟩ **lemma** *abstract-primitive-in-doubt-allow-Deny*:

primitive-matcher-generic $\beta \implies$ *normalized-nnf-match m* \implies
matches (β , *in-doubt-allow*) (*abstract-primitive disc m*) *action.Drop p* \implies
matches (β , *in-doubt-allow*) *m action.Drop p*
 ⟨*proof*⟩ **lemma** *abstract-primitive-in-doubt-allow-Deny2*:

primitive-matcher-generic $\beta \implies$ *normalized-nnf-match m* \implies
 \neg *matches* (β , *in-doubt-allow*) (*abstract-primitive disc m*) *action.Accept p* \implies
 \neg *matches* (β , *in-doubt-allow*) *m action.Accept p*
 ⟨*proof*⟩

theorem *abstract-primitive-in-doubt-allow-generic*:

fixes β ::('i::*len* *common-primitive*, ('i, 'a) *tagged-packet-scheme*) *exact-match-tac*

assumes *generic*: *primitive-matcher-generic* β

and *n*: $\forall r \in \text{set } rs. \text{normalized-nnf-match } (\text{get-match } r)$

and *simple*: *simple-ruleset rs*

defines $\gamma \equiv (\beta, \text{in-doubt-allow})$ **and** *abstract disc* \equiv *optimize-matches* (*abstract-primitive disc*)

shows $\{p. \gamma, p \vdash \langle \text{abstract disc } rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalDeny}\} \subseteq \{p. \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalDeny}\}$
 (is ?deny)
and $\{p. \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow}\} \subseteq \{p. \gamma, p \vdash \langle \text{abstract disc } rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow}\}$
 (is ?allow)
 ⟨proof⟩
corollary *abstract-primitive-in-doubt-allow*:
assumes $\forall r \in \text{set } rs. \text{normalized-nnf-match } (\text{get-match } r)$ **and** *simple-ruleset* rs
defines $\gamma \equiv (\text{common-matcher}, \text{in-doubt-allow})$ **and** *abstract disc* $\equiv \text{optimize-matches } (\text{abstract-primitive disc})$
shows $\{p. \gamma, p \vdash \langle \text{abstract disc } rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalDeny}\} \subseteq \{p. \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalDeny}\}$
and $\{p. \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow}\} \subseteq \{p. \gamma, p \vdash \langle \text{abstract disc } rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow}\}$
 ⟨proof⟩
end

context

begin

private lemma *abstract-primitive-in-doubt-deny-Deny*:

primitive-matcher-generic $\beta \Longrightarrow \text{normalized-nnf-match } m \Longrightarrow$

matches $(\beta, \text{in-doubt-deny}) m \text{ action.Drop } p \Longrightarrow$

matches $(\beta, \text{in-doubt-deny}) (\text{abstract-primitive disc } m) \text{ action.Drop } p$

⟨proof⟩ **lemma** *abstract-primitive-in-doubt-deny-Deny2*:

primitive-matcher-generic $\beta \Longrightarrow \text{normalized-nnf-match } m \Longrightarrow$

$\neg \text{matches } (\beta, \text{in-doubt-deny}) m \text{ action.Accept } p \Longrightarrow$

$\neg \text{matches } (\beta, \text{in-doubt-deny}) (\text{abstract-primitive disc } m) \text{ action.Accept } p$

⟨proof⟩ **lemma** *abstract-primitive-in-doubt-deny-Allow*:

primitive-matcher-generic $\beta \Longrightarrow \text{normalized-nnf-match } m \Longrightarrow$

matches $(\beta, \text{in-doubt-deny}) (\text{abstract-primitive disc } m) \text{ action.Accept } p \Longrightarrow$

matches $(\beta, \text{in-doubt-deny}) m \text{ action.Accept } p$

⟨proof⟩ **lemma** *abstract-primitive-in-doubt-deny-Allow2*:

primitive-matcher-generic $\beta \Longrightarrow \text{normalized-nnf-match } m \Longrightarrow$

$\neg \text{matches } (\beta, \text{in-doubt-deny}) (\text{abstract-primitive disc } m) \text{ action.Drop } p \Longrightarrow$

$\neg \text{matches } (\beta, \text{in-doubt-deny}) m \text{ action.Drop } p$

⟨proof⟩

theorem *abstract-primitive-in-doubt-deny-generic*:

fixes $\beta::('i::\text{len common-primitive}, ('i, 'a) \text{tagged-packet-scheme}) \text{exact-match-tac}$

assumes *generic*: *primitive-matcher-generic* β

and $n::\forall r \in \text{set } rs. \text{normalized-nnf-match } (\text{get-match } r)$

and *simple*: *simple-ruleset* rs

defines $\gamma \equiv (\beta, \text{in-doubt-deny})$ **and** *abstract disc* $\equiv \text{optimize-matches } (\text{abstract-primitive disc})$

shows $\{p. \gamma, p \vdash \langle \text{abstract disc } rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow}\} \subseteq \{p. \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow}\}$

```

      (is ?allow)
    and {p.  $\gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Decision FinalDeny\} \subseteq \{p. \gamma, p \vdash \langle abstract
disc rs, Undecided \rangle \Rightarrow_{\alpha} Decision FinalDeny\}
      (is ?deny)
    <proof>
  end$ 
```

end

41 Iptables to Simple Firewall and Vice Versa

```

theory SimpleFw-Compliance
imports Simple-Firewall.SimpleFw-Semantics
        ../Primitive-Matchers/Transform
        ../Primitive-Matchers/Primitive-Abstract
begin

```

41.1 Simple Match to MatchExpr

```

fun simple-match-to-ipportiface-match :: 'i::len simple-match  $\Rightarrow$  'i common-primitive
match-expr where
  simple-match-to-ipportiface-match (|iiface=iif, oiface=oif, src=sip, dst=dip, proto=p,
sports=sps, dports=dps |) =
    MatchAnd (Match (Iiface iif)) (MatchAnd (Match (Oiface oif))
      (MatchAnd (Match (Src (uncurry IpAddrNetmask sip)))
        (MatchAnd (Match (Dst (uncurry IpAddrNetmask dip)))
          (case p of ProtoAny  $\Rightarrow$  MatchAny
            | Proto prim-p  $\Rightarrow$ 
              (MatchAnd (Match (Prot p))
                (MatchAnd (Match (Src-Ports (L4Ports prim-p [sps])))
                  (Match (Dst-Ports (L4Ports prim-p [dps])))
                ))
          )))
    ))))

```

```

lemma ports-to-set-singleton-simple-match-port: p  $\in$  ports-to-set [a]  $\longleftrightarrow$  simple-match-port
a p
  <proof>

```

```

theorem simple-match-to-ipportiface-match-correct:
  assumes valid: simple-match-valid sm
  shows matches (common-matcher,  $\alpha$ ) (simple-match-to-ipportiface-match sm) a
p  $\longleftrightarrow$  simple-matches sm p
  <proof>

```

41.2 MatchExpr to Simple Match

fun *common-primitive-match-to-simple-match* :: 'i::len *common-primitive match-expr*
 \Rightarrow 'i *simple-match option* **where**

```

common-primitive-match-to-simple-match MatchAny = Some (simple-match-any)
|
  common-primitive-match-to-simple-match (MatchNot MatchAny) = None |
  common-primitive-match-to-simple-match (Match (Iiface iif)) = Some (simple-match-any(
iiface := iif )) |
  common-primitive-match-to-simple-match (Match (Oiface oif)) = Some (simple-match-any(
oiface := oif )) |
  common-primitive-match-to-simple-match (Match (Src (IpAddrNetmask pre len)))
= Some (simple-match-any( src := (pre, len) )) |
  common-primitive-match-to-simple-match (Match (Dst (IpAddrNetmask pre len)))
= Some (simple-match-any( dst := (pre, len) )) |
  common-primitive-match-to-simple-match (Match (Prot p)) = Some (simple-match-any(
proto := p )) |
  common-primitive-match-to-simple-match (Match (Src-Ports (L4Ports p []))) =
None |
  common-primitive-match-to-simple-match (Match (Src-Ports (L4Ports p [(s,e)])))
= Some (simple-match-any( proto := Proto p, sports := (s,e) )) |
  common-primitive-match-to-simple-match (Match (Dst-Ports (L4Ports p []))) =
None |
  common-primitive-match-to-simple-match (Match (Dst-Ports (L4Ports p [(s,e)])))
= Some (simple-match-any( proto := Proto p, dports := (s,e) )) |
  common-primitive-match-to-simple-match (MatchNot (Match (Prot ProtoAny)))
= None |
  common-primitive-match-to-simple-match (MatchAnd m1 m2) = (case (common-primitive-match-to-simple-m
m1, common-primitive-match-to-simple-match m2) of
    (None, -)  $\Rightarrow$  None
  | (-, None)  $\Rightarrow$  None
  | (Some m1', Some m2')  $\Rightarrow$  simple-match-and m1' m2') |
  — undefined cases, normalize before!
  common-primitive-match-to-simple-match (Match (Src (IpAddr -))) = undefined
|
  common-primitive-match-to-simple-match (Match (Src (IpAddrRange - -))) =
undefined |
  common-primitive-match-to-simple-match (Match (Dst (IpAddr -))) = undefined
|
  common-primitive-match-to-simple-match (Match (Dst (IpAddrRange - -))) =
undefined |
  common-primitive-match-to-simple-match (MatchNot (Match (Prot -))) = unde-
fined |
  common-primitive-match-to-simple-match (MatchNot (Match (Iiface -))) = unde-
fined |
  common-primitive-match-to-simple-match (MatchNot (Match (Oiface -))) = unde-
fined |
  common-primitive-match-to-simple-match (MatchNot (Match (Src -))) = unde-
fined |
  common-primitive-match-to-simple-match (MatchNot (Match (Dst -))) = unde-

```

```

fined |
  common-primitive-match-to-simple-match (MatchNot (MatchAnd - -)) = unde-
fined |
  common-primitive-match-to-simple-match (MatchNot (MatchNot -)) = undefined
|
  common-primitive-match-to-simple-match (Match (Src-Ports -)) = undefined |
  common-primitive-match-to-simple-match (Match (Dst-Ports -)) = undefined |
  common-primitive-match-to-simple-match (MatchNot (Match (Src-Ports -))) =
undefined |
  common-primitive-match-to-simple-match (MatchNot (Match (Dst-Ports -))) =
undefined |
  common-primitive-match-to-simple-match (Match (CT-State -)) = undefined |
  common-primitive-match-to-simple-match (Match (L4-Flags -)) = undefined |
  common-primitive-match-to-simple-match (MatchNot (Match (L4-Flags -))) =
undefined |
  common-primitive-match-to-simple-match (Match (Extra -)) = undefined |
  common-primitive-match-to-simple-match (MatchNot (Match (Extra -))) = un-
defined |
  common-primitive-match-to-simple-match (MatchNot (Match (CT-State -))) =
undefined

```

41.2.1 Normalizing Interfaces

As for now, negated interfaces are simply not allowed

definition *normalized-ifaces* :: 'i::len common-primitive match-expr ⇒ bool **where**
normalized-ifaces m ≡ ¬ has-disc-negated (λa. is-Iiface a ∨ is-Oiface a) False m

41.2.2 Normalizing Protocols

As for now, negated protocols are simply not allowed

definition *normalized-protocols* :: 'i::len common-primitive match-expr ⇒ bool **where**
normalized-protocols m ≡ ¬ has-disc-negated is-Prot False m

lemma *match-iface-simple-match-any-simps*:

```

  match-iface (iface simple-match-any) (p-iface p)
  match-iface (oiface simple-match-any) (p-oiface p)
  simple-match-ip (src simple-match-any) (p-src p)
  simple-match-ip (dst simple-match-any) (p-dst p)
  match-proto (proto simple-match-any) (p-proto p)
  simple-match-port (sports simple-match-any) (p-sport p)
  simple-match-port (dports simple-match-any) (p-dport p)
  ⟨proof⟩

```

theorem *common-primitive-match-to-simple-match*:

assumes *normalized-src-ports m*
and *normalized-dst-ports m*
and *normalized-src-ips m*
and *normalized-dst-ips m*
and *normalized-ifaces m*
and *normalized-protocols m*
and \neg *has-disc is-L4-Flags m*
and \neg *has-disc is-CT-State m*
and \neg *has-disc is-MultiportPorts m*
and \neg *has-disc is-Extra m*
shows (*Some sm = common-primitive-match-to-simple-match m* \longrightarrow *matches*
(common-matcher, α) m a p \longleftrightarrow *simple-matches sm p*) \wedge
(common-primitive-match-to-simple-match m = None \longrightarrow \neg *matches*
(common-matcher, α) m a p)
 {*proof*}

lemma *simple-fw-remdups-Rev: simple-fw (remdups-rev rs) p = simple-fw rs p*
 {*proof*}

fun *action-to-simple-action* :: *action* \Rightarrow *simple-action* **where**
action-to-simple-action action.Accept = simple-action.Accept |
action-to-simple-action action.Drop = simple-action.Drop |
action-to-simple-action - = undefined

definition *check-simple-fw-preconditions* :: '*i::len common-primitive rule list* \Rightarrow
bool **where**

check-simple-fw-preconditions rs \equiv $\forall r \in$ *set rs. (case r of (Rule m a) \Rightarrow*
normalized-src-ports m \wedge
normalized-dst-ports m \wedge
normalized-src-ips m \wedge
normalized-dst-ips m \wedge
normalized-ifaces m \wedge
normalized-protocols m \wedge
 \neg *has-disc is-L4-Flags m* \wedge
 \neg *has-disc is-CT-State m* \wedge
 \neg *has-disc is-MultiportPorts m* \wedge
 \neg *has-disc is-Extra m* \wedge
(a = action.Accept \vee a = action.Drop))

lemma *normalized-src-ports m* \Longrightarrow *normalized-nnf-match m*
 {*proof*}

lemma \neg *matcheq-matchNone m* \Longrightarrow *normalized-src-ports m* \Longrightarrow *normalized-nnf-match*
m
 {*proof*}

value *check-simple-fw-preconditions* [*Rule (MatchNot (MatchNot (MatchNot (Match*
(Src a)))) action.Accept]

definition *to-simple-firewall* :: 'i::len common-primitive rule list \Rightarrow 'i simple-rule list **where**

to-simple-firewall rs \equiv if *check-simple-fw-preconditions* rs then

List.map-filter (λr . case r of Rule m a \Rightarrow

(case (*common-primitive-match-to-simple-match* m) of None \Rightarrow None |

Some sm \Rightarrow Some (*SimpleRule* sm (*action-to-simple-action* a)))) rs

else undefined

lemma *to-simple-firewall-simps*:

to-simple-firewall [] = []

check-simple-fw-preconditions ((Rule m a)#rs) \Longrightarrow *to-simple-firewall* ((Rule m a)#rs) = (case *common-primitive-match-to-simple-match* m of

None \Rightarrow *to-simple-firewall* rs

| Some sm \Rightarrow (*SimpleRule* sm (*action-to-simple-action* a)) # *to-simple-firewall* rs)

\neg *check-simple-fw-preconditions* rs' \Longrightarrow *to-simple-firewall* rs' = undefined

\langle proof \rangle

lemma *check-simple-fw-preconditions*

[Rule (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (127, 0, 0, 0)) 8)))

(*MatchAnd* (*Match* (*Dst-Ports* (*L4Ports* TCP [(0, 65535)])))
(*Match* (*Src-Ports* (*L4Ports* TCP [(0, 65535)])))))]

Drop] \langle proof \rangle

lemma *to-simple-firewall*

[Rule (*MatchAnd* (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (127, 0, 0, 0)) 8)))

(*MatchAnd* (*Match* (*Dst-Ports* (*L4Ports* TCP [(0, 65535)])))
(*Match* (*Src-Ports* (*L4Ports* TCP [(0, 65535)])))))]

Drop] =

[*SimpleRule*

(*iiface* = *Iface* "+", *oiface* = *Iface* "+", *src* = (0x7F000000, 8), *dst* = (0, 0),
proto = *Proto* 6, *sports* = (0, 0xFFFF),

dports = (0, 0xFFFF))]

simple-action.Drop] \langle proof \rangle

lemma *check-simple-fw-preconditions* [Rule (*MatchAnd* *MatchAny* *MatchAny*) *Drop*] \langle proof \rangle

lemma *to-simple-firewall* [Rule (*MatchAnd* *MatchAny* (*MatchAny*::32 *common-primitive match-expr*)) *Drop*] =

[*SimpleRule*

(*iiface* = *Iface* "+", *oiface* = *Iface* "+", *src* = (0, 0), *dst* = (0, 0), *proto* =
ProtoAny, *sports* = (0, 0xFFFF),

dports = (0, 0xFFFF))]

simple-action.Drop] \langle proof \rangle

lemma *to-simple-firewall* [Rule (*Match* (*Src* (*IpAddrNetmask* (*ipv4addr-of-dotdecimal* (127, 0, 0, 0)) 8))] *Drop*] =

[*SimpleRule*
 (iface = Iface "+", oiface = Iface "+", src = (0x7F000000, 8), dst = (0, 0),
 proto = ProtoAny, sports = (0, 0xFFFF),
 dports = (0, 0xFFFF))
 simple-action.Drop] <proof>

theorem *to-simple-firewall*: check-simple-fw-preconditions rs \implies approximating-bigstep-fun
 (common-matcher, α) p rs Undecided = simple-fw (to-simple-firewall rs) p
 <proof>

lemma *ctstate-assume-new-not-has-CT-State*:

$r \in \text{set } (\text{ctstate-assume-new } rs) \implies \neg \text{has-disc is-CT-State } (\text{get-match } r)$
 <proof>

The precondition for the simple firewall can be easily fulfilled. The subset relation is due to abstracting over some primitives (e.g., negated primitives, 14 flags)

theorem *transform-simple-fw-upper*:

defines *preprocess rs* \equiv upper-closure (optimize-matches abstract-for-simple-firewall
 (upper-closure (packet-assume-new rs)))

and *newpkt p* \equiv match-tcp-flags ipt-tcp-syn (p-tcp-flags p) \wedge p-tag-ctstate p =
 CT-New

assumes *simplers*: simple-ruleset (rs:: 'i::len common-primitive rule list)

— the preconditions for the simple firewall are fulfilled, definitely no runtime failure

shows *check-simple-fw-preconditions* (preprocess rs)

— the set of new packets, which are accepted is an overapproximations

and {p. (common-matcher, in-doubt-allow), p \vdash (rs, Undecided) \Rightarrow_{α} Decision FinalAllow \wedge newpkt p} \subseteq

{p. simple-fw (to-simple-firewall (preprocess rs)) p = Decision FinalAllow \wedge newpkt p}

— Fun fact: The theorem holds for a tagged packet. The simple firewall just ignores the tag. You may explicitly untag, if you wish to, but a 'i tagged-packet is just an extension of the 'i simple-packet used by the simple firewall

<proof>

theorem *transform-simple-fw-lower*:

defines *preprocess rs* \equiv lower-closure (optimize-matches abstract-for-simple-firewall
 (lower-closure (packet-assume-new rs)))

and *newpkt p* \equiv match-tcp-flags ipt-tcp-syn (p-tcp-flags p) \wedge p-tag-ctstate p =
 CT-New

assumes *simplers*: simple-ruleset (rs:: 'i::len common-primitive rule list)

— the preconditions for the simple firewall are fulfilled, definitely no runtime failure

shows *check-simple-fw-preconditions* (*preprocess rs*)
 — the set of new packets, which are accepted is an underapproximation
and $\{p. \text{simple-fw } (to\text{-simple-firewall } (preprocess\ rs))\ p = Decision\ FinalAllow \wedge newpkt\ p\} \subseteq$
 $\{p. (common\ matcher, in\ doubt\ deny), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Decision\ FinalAllow \wedge newpkt\ p\}$
 $\langle proof \rangle$

definition *to-simple-firewall-without-interfaces ipassmt rtblo rs* \equiv
to-simple-firewall
(upper-closure
(optimize-matches (abstract-primitive ($\lambda r. case\ r\ of\ Pos\ a \Rightarrow is\ Iiface\ a \vee is\ Oiface\ a \mid Neg\ a \Rightarrow is\ Iiface\ a \vee is\ Oiface\ a$)))
(optimize-matches abstract-for-simple-firewall
(upper-closure
(iface-try-rewrite ipassmt rtblo
(upper-closure
(packet-assume-new rs))))))

theorem *to-simple-firewall-without-interfaces:*

defines *newpkt p* $\equiv match\ tcp\ flags\ ipt\ tcp\ syn\ (p\ tcp\ flags\ p) \wedge p\ tag\ ctstate\ p = CT\ New$

assumes *simplers: simple-ruleset (rs:: 'i::len common-primitive rule list)*

— well-formed *ipassmt*

and *wf-ipassmt1: ipassmt-sanity-nowildcards (map-of ipassmt)* **and** *wf-ipassmt2: distinct (map fst ipassmt)*

— There are no spoofed packets (probably by kernel's reverse path filter or our checker). This assumption implies that *ipassmt* lists ALL interfaces (!!).

and *nospoofing: $\forall (p::('i::len, 'a) tagged\ packet\ scheme).$*

$\exists ips. (map\ of\ ipassmt)\ (Iface\ (p\ iiface\ p)) = Some\ ips \wedge p\ src\ p \in ipcidr\ union\ set\ (set\ ips)$

— If a routing table was passed, the output interface for any packet we consider is decided based on it.

and *routing-decided: $\bigwedge rtbl\ (p::('i, 'a) tagged\ packet\ scheme). rtblo = Some\ rtbl \Rightarrow output\ iface\ (routing\ table\ semantics\ rtbl\ (p\ dst\ p)) = p\ oiface\ p$*

— A passed routing table is wellformed

and *correct-routing: $\bigwedge rtbl. rtblo = Some\ rtbl \Rightarrow correct\ routing\ rtbl$*

— A passed routing table contains no interfaces with wildcard names

and *routing-no-wildcards: $\bigwedge rtbl. rtblo = Some\ rtbl \Rightarrow ipassmt\ sanity\ nowildcards\ (map\ of\ (routing\ ipassmt\ rtbl))$*

— the set of new packets, which are accepted is an overapproximations

shows $\{p::('i, 'a) tagged\ packet\ scheme. (common\ matcher, in\ doubt\ allow), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Decision\ FinalAllow \wedge newpkt\ p\} \subseteq$

$\{p::('i, 'a) \text{ tagged-packet-scheme. simple-fw (to-simple-firewall-without-interfaces ipassmt rtblo rs) } p = \text{Decision FinalAllow} \wedge \text{newpkt } p\}$

and $\forall r \in \text{set (to-simple-firewall-without-interfaces ipassmt rtblo rs)}$.
 $\text{iiface (match-sel } r) = \text{ifaceAny} \wedge \text{oiface (match-sel } r) = \text{ifaceAny}$
 $\langle \text{proof} \rangle$

end

theory *Semantics-Embeddings*

imports *Simple-Firewall/SimpleFw-Compliance Matching-Embeddings Semantics Semantics-Ternary/Semantics-Ternary*

begin

42 Semantics Embedding

42.1 Tactic *in-doubt-allow*

lemma *iptables-bigstep-undecided-to-undecided-in-doubt-allow-approx*:

assumes *agree: matcher-agree-on-exact-matches* $\gamma \beta$
and good: *good-ruleset* rs **and semantics:** $\Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Undecided}$
shows $(\beta, \text{in-doubt-allow}), p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Undecided} \vee (\beta, \text{in-doubt-allow}), p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow}$
 $\langle \text{proof} \rangle$

lemma *FinalAllow-approximating-in-doubt-allow*:

assumes *agree: matcher-agree-on-exact-matches* $\gamma \beta$
and good: *good-ruleset* rs **and semantics:** $\Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Decision FinalAllow}$
shows $(\beta, \text{in-doubt-allow}), p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow}$
 $\langle \text{proof} \rangle$

corollary *FinalAllows-subseteq-in-doubt-allow: matcher-agree-on-exact-matches* γ
 $\beta \Rightarrow \text{good-ruleset } rs \Rightarrow$

$\{p. \Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Decision FinalAllow}\} \subseteq \{p. (\beta, \text{in-doubt-allow}), p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow}\}$
 $\langle \text{proof} \rangle$

corollary *new-packets-to-simple-firewall-overapproximation*:

defines *preprocess* $rs \equiv \text{upper-closure (optimize-matches abstract-for-simple-firewall (upper-closure (packet-assume-new } rs))}$

and *newpkt* $p \equiv \text{match-tcp-flags ipt-tcp-syn (p-tcp-flags } p) \wedge \text{p-tag-ctstate } p = \text{CT-New}$

fixes $p :: ('i::\text{len}, 'pkt\text{-ext}) \text{ tagged-packet-scheme}$

assumes *matcher-agree-on-exact-matches* γ *common-matcher* **and** *simple-ruleset*

rs

shows $\{p. \Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalAllow \wedge newpkt\ p\} \subseteq \{p. simple-fw\ (to-simple-firewall\ (preprocess\ rs))\ p = Decision\ FinalAllow \wedge newpkt\ p\}$
 $\langle proof \rangle$

lemma *approximating-bigstep-undecided-to-undecided-in-doubt-allow-approx: matcher-agree-on-exact-matches*

$\gamma\ \beta \Longrightarrow$
good-ruleset $rs \Longrightarrow$
 $(\beta, in-doubt-allow), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Undecided \Longrightarrow \Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Undecided \vee \Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalDeny$
 $\langle proof \rangle$

lemma *FinalDeny-approximating-in-doubt-allow: matcher-agree-on-exact-matches*

$\gamma\ \beta \Longrightarrow$
good-ruleset $rs \Longrightarrow$
 $(\beta, in-doubt-allow), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Decision\ FinalDeny \Longrightarrow \Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalDeny$
 $\langle proof \rangle$

corollary *FinalDenys-subseteq-in-doubt-allow: matcher-agree-on-exact-matches*

$\gamma\ \beta \Longrightarrow$ *good-ruleset* $rs \Longrightarrow$
 $\{p. (\beta, in-doubt-allow), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Decision\ FinalDeny\} \subseteq \{p. \Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalDeny\}$
 $\langle proof \rangle$

If our approximating firewall (the executable version) concludes that we deny a packet, the exact semantic agrees that this packet is definitely denied!

corollary *matcher-agree-on-exact-matches* $\gamma\ \beta \Longrightarrow$ *good-ruleset* $rs \Longrightarrow$

approximating-bigstep-fun $(\beta, in-doubt-allow)\ p\ rs\ Undecided = (Decision\ FinalDeny) \Longrightarrow \Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalDeny$
 $\langle proof \rangle$

42.2 Tactic *in-doubt-deny*

lemma *iptables-bigstep-undecided-to-undecided-in-doubt-deny-approx: matcher-agree-on-exact-matches*

$\gamma\ \beta \Longrightarrow$
good-ruleset $rs \Longrightarrow$
 $\Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Undecided \Longrightarrow$
 $(\beta, in-doubt-deny), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Undecided \vee (\beta, in-doubt-deny), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Decision\ FinalDeny$
 $\langle proof \rangle$

lemma *FinalDeny-approximating-in-doubt-deny: matcher-agree-on-exact-matches*

$\gamma\ \beta \Longrightarrow$

$good\text{-ruleset } rs \implies$
 $\Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalDeny \implies (\beta, in\text{-doubt-deny}), p \vdash \langle rs,$
 $Undecided \rangle \Rightarrow_{\alpha} Decision\ FinalDeny$
 $\langle proof \rangle$

lemma *approximating-bigstep-undecided-to-undecided-in-doubt-deny-approx: matcher-agree-on-exact-matches*
 $\gamma \beta \implies$
 $good\text{-ruleset } rs \implies$
 $(\beta, in\text{-doubt-deny}), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Undecided \implies \Gamma, \gamma, p \vdash \langle rs, Unde-$
 $cided \rangle \Rightarrow Undecided \vee \Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalAllow$
 $\langle proof \rangle$

lemma *FinalAllow-approximating-in-doubt-deny: matcher-agree-on-exact-matches*
 $\gamma \beta \implies$
 $good\text{-ruleset } rs \implies$
 $(\beta, in\text{-doubt-deny}), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Decision\ FinalAllow \implies \Gamma, \gamma, p \vdash \langle rs,$
 $Undecided \rangle \Rightarrow Decision\ FinalAllow$
 $\langle proof \rangle$

corollary *FinalAllows-subseteq-in-doubt-deny: matcher-agree-on-exact-matches* γ
 $\beta \implies good\text{-ruleset } rs \implies$
 $\{p. (\beta, in\text{-doubt-deny}), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Decision\ FinalAllow\} \subseteq \{p. \Gamma, \gamma, p \vdash$
 $\langle rs, Undecided \rangle \Rightarrow Decision\ FinalAllow\}$
 $\langle proof \rangle$

corollary *new-packets-to-simple-firewall-underapproximation:*
defines $preprocess\ rs \equiv lower\text{-closure } (optimize\text{-matches } abstract\text{-for-simple-firewall } (lower\text{-closure } (packet\text{-assume-new } rs)))$
and $newpkt\ p \equiv match\text{-tcp-flags } ipt\text{-tcp-syn } (p\text{-tcp-flags } p) \wedge p\text{-tag-ctstate } p = CT\text{-New}$
fixes $p :: ('i::len, 'pkt\text{-ext})\ tagged\text{-packet-scheme}$
assumes *matcher-agree-on-exact-matches* γ *common-matcher* **and** *simple-ruleset* rs
shows $\{p. simple\text{-fw } (to\text{-simple-firewall } (preprocess\ rs))\ p = Decision\ FinalAllow \wedge newpkt\ p\} \subseteq \{p. \Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalAllow \wedge newpkt\ p\}$
 $\langle proof \rangle$

42.3 Approximating Closures

theorem *FinalAllowClosure:*
assumes *matcher-agree-on-exact-matches* γ β **and** *good-ruleset* rs
shows $\{p. (\beta, in\text{-doubt-deny}), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Decision\ FinalAllow\} \subseteq$
 $\{p. \Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalAllow\}$

and $\{p. \Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalAllow\} \subseteq \{p. (\beta, in-doubt-allow), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Decision\ FinalAllow\}$
 $\langle proof \rangle$

theorem *FinalDenyClosure*:

assumes *matcher-agree-on-exact-matches* γ β **and** *good-ruleset* rs
shows $\{p. (\beta, in-doubt-allow), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Decision\ FinalDeny\} \subseteq$
 $\{p. \Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalDeny\}$
and $\{p. \Gamma, \gamma, p \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalDeny\} \subseteq \{p. (\beta, in-doubt-deny), p \vdash \langle rs, Undecided \rangle \Rightarrow_{\alpha} Decision\ FinalDeny\}$
 $\langle proof \rangle$

42.4 Exact Embedding

lemma *LukassLemma*: **assumes** *agree*: *matcher-agree-on-exact-matches* γ β

and *noUnknown*: $(\forall r \in set\ rs. ternary-ternary-eval\ (map-match-tac\ \beta\ p\ (get-match\ r)) \neq TernaryUnknown)$

and *good*: *good-ruleset* rs

shows $(\beta, \alpha), p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t \iff \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$
 $\langle proof \rangle$

For rulesets without *Calls*, the approximating ternary semantics can perfectly simulate the Boolean semantics.

theorem *β_{magic} -approximating-bigstep-iff-iptables-bigstep*:

assumes $\forall r \in set\ rs. \forall c. get-action\ r \neq Call\ c$

shows $((\beta_{magic}\ \gamma), \alpha), p \vdash \langle rs, s \rangle \Rightarrow_{\alpha} t \iff \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$
 $\langle proof \rangle$

corollary *β_{magic} -approximating-bigstep-fun-iff-iptables-bigstep*:

assumes *good-ruleset* rs

shows *approximating-bigstep-fun* $(\beta_{magic}\ \gamma, \alpha)\ p\ rs\ s = t \iff \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$
 $\langle proof \rangle$

The function *optimize-primitive-univ* was only applied to the ternary semantics. It is, in fact, also correct for the Boolean semantics, assuming the *common-matcher*.

lemma *Semantics-optimize-primitive-univ-common-matcher*:

assumes *matcher-agree-on-exact-matches* γ *common-matcher*

shows *Semantics.matches* γ $(optimize-primitive-univ\ m)\ p = Semantics.matches\ \gamma\ m\ p$
 $\langle proof \rangle$

end

theory *Iptables-Semantics*

imports *Semantics-Embeddings Semantics-Ternary/Normalized-Matches*

begin

43 Normalizing Rulesets in the Boolean Big Step Semantics

corollary *normalize-rules-dnf-correct-BooleanSemantics:*

assumes *good-ruleset rs*

shows $\Gamma, \gamma, p \vdash \langle \text{normalize-rules-dnf } rs, s \rangle \Rightarrow t \iff \Gamma, \gamma, p \vdash \langle rs, s \rangle \Rightarrow t$
<proof>

end

theory *Code-Interface*

imports

Common-Primitive-toString

IP-Addresses.IP-Address-Parser

../Call-Return-Unfolding

Transform

No-Spoof

../Simple-Firewall/SimpleFw-Compliance

Simple-Firewall.SimpleFw-toString

Simple-Firewall.Service-Matrix

../Semantics-Ternary/Optimizing

../Semantics-Goto

HOL-Library.Code-Target-Nat

HOL-Library.Code-Target-Int

Native-Word.Code-Target-Int-Bit

begin

44 Code Interface

HACK: rewrite quotes such that they are better printable by Isabelle

definition *quote-rewrite* :: *string* \Rightarrow *string* **where**

quote-rewrite \equiv *map* ($\lambda c.$ *if* *c* = *char-of-nat 34* *then* *CHR* *"~"* *else* *c*)

lemma *quote-rewrite* (*"foo"*@[*char-of-nat 34*]) = *"foo~"* *<proof>*

The parser returns the *'i* *common-primitive ruleset* not as a map but as an association list. This function converts it

definition *map-of-string-ipv4*

:: (*string* \times *32* *common-primitive rule list*) *list* \Rightarrow *string* \rightarrow *32* *common-primitive rule list* **where**

map-of-string-ipv4 *rs* = *map-of* *rs*

definition *map-of-string-ipv6*

:: (*string* \times *128* *common-primitive rule list*) *list* \Rightarrow *string* \rightarrow *128* *common-primitive rule list* **where**

map-of-string-ipv6 *rs* = *map-of* *rs*

definition *map-of-string*

:: (*string* \times *'i* *common-primitive rule list*) *list* \Rightarrow *string* \rightarrow *'i* *common-primitive rule list* **where**

map-of-string rs = map-of rs

definition *unfold-ruleset-CHAIN-safe* :: *string* \Rightarrow *action* \Rightarrow '*i*::*len* *common-primitive ruleset* \Rightarrow '*i* *common-primitive rule list option* **where**
unfold-ruleset-CHAIN-safe = *unfold-optimize-ruleset-CHAIN optimize-primitive-univ*

lemma (*unfold-ruleset-CHAIN-safe chain a rs = Some rs'*) \implies *simple-ruleset rs'*
{*proof*}

definition *unfold-ruleset-CHAIN* :: *string* \Rightarrow *action* \Rightarrow '*i*::*len* *common-primitive ruleset* \Rightarrow '*i* *common-primitive rule list* **where**
unfold-ruleset-CHAIN chain default-action rs = the (unfold-ruleset-CHAIN-safe chain default-action rs)

definition *unfold-ruleset-FORWARD* :: *action* \Rightarrow '*i*::*len* *common-primitive ruleset* \Rightarrow '*i*::*len* *common-primitive rule list* **where**
unfold-ruleset-FORWARD = *unfold-ruleset-CHAIN "FORWARD"*

definition *unfold-ruleset-INPUT* :: *action* \Rightarrow '*i*::*len* *common-primitive ruleset* \Rightarrow '*i*::*len* *common-primitive rule list* **where**
unfold-ruleset-INPUT = *unfold-ruleset-CHAIN "INPUT"*

definition *unfold-ruleset-OUTPUT* :: *action* \Rightarrow '*i*::*len* *common-primitive ruleset* \Rightarrow '*i*::*len* *common-primitive rule list* **where**
unfold-ruleset-OUTPUT \equiv *unfold-ruleset-CHAIN "OUTPUT"*

lemma *let fw = ["FORWARD" \mapsto []] in*
unfold-ruleset-FORWARD action.Drop fw
= [*Rule (MatchAny :: 32 common-primitive match-expr) action.Drop*] {*proof*}

definition *nat-to-8word* :: *nat* \Rightarrow *8 word* **where**
nat-to-8word i \equiv *of-nat i*

definition *nat-to-16word* :: *nat* \Rightarrow *16 word* **where**
nat-to-16word i \equiv *of-nat i*

definition *integer-to-16word* :: *integer* \Rightarrow *16 word* **where**
integer-to-16word i \equiv *nat-to-16word (nat-of-integer i)*

context

```

begin
  private definition is-pos-Extra :: 'i::len common-primitive negation-type ⇒ bool
  where
    is-pos-Extra a ≡ (case a of Pos (Extra -) ⇒ True | - ⇒ False)
  private definition get-pos-Extra :: 'i::len common-primitive negation-type ⇒
  string where
    get-pos-Extra a ≡ (case a of Pos (Extra e) ⇒ e | - ⇒ undefined)

  fun compress-parsed-extra
    :: 'i::len common-primitive negation-type list ⇒ 'i common-primitive nega-
  tion-type list where
    compress-parsed-extra [] = [] |
    compress-parsed-extra (a1#a2#as) = (if is-pos-Extra a1 ∧ is-pos-Extra a2
    then compress-parsed-extra (Pos (Extra (get-pos-Extra a1@''@get-pos-Extra
  a2))#as)
    else a1#compress-parsed-extra (a2#as)
    ) |
    compress-parsed-extra (a#as) = a#compress-parsed-extra as

  lemma compress-parsed-extra
    (map Pos [Extra "--m", (Extra "recent" :: 32 common-primitive),
    Extra "--update", Extra "--seconds", Extra "60",
    Iiface (Iface "foobar"),
    Extra "--name", Extra "DEFAULT", Extra "--resource"]) =
  map Pos [Extra "--m recent --update --seconds 60",
    Iiface (Iface "foobar"),
    Extra "--name DEFAULT --resource"] <proof> lemma eval-ternary-And-Unknown-Unknown:
  eval-ternary-And TernaryUnknown (eval-ternary-And TernaryUnknown tv) =
  eval-ternary-And TernaryUnknown tv
  <proof> lemma is-pos-Extra-alist-and:
  is-pos-Extra a ⇒ alist-and (a#as) = MatchAnd (Match (Extra (get-pos-Extra
  a))) (alist-and as)
  <proof> lemma compress-parsed-extra-matchexpr-helper:
  ternary-ternary-eval (map-match-tac common-matcher p (alist-and (compress-parsed-extra
  as))) =
  ternary-ternary-eval (map-match-tac common-matcher p (alist-and as))
  <proof>

```

This lemma justifies that it is okay to fold together the parsed unknown tokens

```

lemma compress-parsed-extra-matchexpr:
  matches (common-matcher, α) (alist-and (compress-parsed-extra as)) =
  matches (common-matcher, α) (alist-and as)
  <proof>
end

```

44.1 L4 Ports Parser Helper

context

begin

Replace all matches on ports with the unspecified 0 protocol with the given *primitive-protocol*.

private definition *fill-l4-protocol-raw*

$\text{:: } \text{primitive-protocol} \Rightarrow 'i::\text{len common-primitive negation-type list} \Rightarrow 'i \text{ common-primitive negation-type list}$

where

$\text{fill-l4-protocol-raw protocol} \equiv \text{NegPos-map}$

$(\lambda m. \text{case } m \text{ of Src-Ports (L4Ports } x \text{ pts)} \Rightarrow \text{if } x \neq 0 \text{ then undefined else Src-Ports (L4Ports protocol pts)}$

$\quad | \text{Dst-Ports (L4Ports } x \text{ pts)} \Rightarrow \text{if } x \neq 0 \text{ then undefined else Dst-Ports (L4Ports protocol pts)}$

$\quad | \text{MultiportPorts (L4Ports } x \text{ pts)} \Rightarrow \text{if } x \neq 0 \text{ then undefined else MultiportPorts (L4Ports protocol pts)}$

$\quad | \text{Prot } - \Rightarrow \text{undefined}$ — there should be no more match on the protocol if it was parsed from an iptables-save line

$\quad | m \Rightarrow m$

)

lemma *fill-l4-protocol-raw TCP* [Neg (Dst (IpAddrNetmask (ipv4addr-of-dotdecimal (127, 0, 0, 0)) 8)), Pos (Src-Ports (L4Ports 0 [(22,22)]))] =

[Neg (Dst (IpAddrNetmask 0x7F000000 8)), Pos (Src-Ports (L4Ports 6 [(0x16, 0x16)]))] <proof>

fun *fill-l4-protocol*

$\text{:: } 'i::\text{len common-primitive negation-type list} \Rightarrow 'i::\text{len common-primitive negation-type list}$

where

$\text{fill-l4-protocol } [] = []$ |

$\text{fill-l4-protocol (Pos (Prot (Proto protocol)) \# ms)} = \text{Pos (Prot (Proto protocol))}$

$\# \text{fill-l4-protocol-raw protocol ms}$ |

$\text{fill-l4-protocol (Pos (Src-Ports -) \# -)} = \text{undefined}$ |

$\text{fill-l4-protocol (Pos (Dst-Ports -) \# -)} = \text{undefined}$ |

$\text{fill-l4-protocol (Pos (MultiportPorts -) \# -)} = \text{undefined}$ |

$\text{fill-l4-protocol (Neg (Src-Ports -) \# -)} = \text{undefined}$ |

$\text{fill-l4-protocol (Neg (Dst-Ports -) \# -)} = \text{undefined}$ |

$\text{fill-l4-protocol (Neg (MultiportPorts -) \# -)} = \text{undefined}$ |

$\text{fill-l4-protocol (m \# ms)} = m \# \text{fill-l4-protocol ms}$

lemma *fill-l4-protocol* [Neg (Dst (IpAddrNetmask (ipv4addr-of-dotdecimal (127, 0, 0, 0)) 8))

, Neg (Prot (Proto UDP))

, Pos (Src (IpAddrNetmask (ipv4addr-of-dotdecimal (127, 0, 0, 0)) 8))

, Pos (Prot (Proto TCP))

, Pos (Extra "foo")

, Pos (Src-Ports (L4Ports 0 [(22,22)]))

, Neg (Extra "Bar")] =

```

[ Neg (Dst (IpAddrNetmask 0x7F000000 8))
, Neg (Prot (Proto UDP))
, Pos (Src (IpAddrNetmask 0x7F000000 8))
, Pos (Prot (Proto TCP))
, Pos (Extra "foo")
, Pos (Src-Ports (L4Ports TCP [(0x16, 0x16)]))
, Neg (Extra "Bar")] <proof>
end

```

definition *prefix-to-strange-inverse-cisco-mask*:: $nat \Rightarrow (nat \times nat \times nat \times nat)$
where
prefix-to-strange-inverse-cisco-mask $n \equiv \text{dotdecimal-of-ipv4addr } (Bit-Operations.not$
 $(mask\ n \ll 32 - n))$

lemma *prefix-to-strange-inverse-cisco-mask* 8 = (0, 255, 255, 255) <proof>
lemma *prefix-to-strange-inverse-cisco-mask* 16 = (0, 0, 255, 255) <proof>
lemma *prefix-to-strange-inverse-cisco-mask* 24 = (0, 0, 0, 255) <proof>
lemma *prefix-to-strange-inverse-cisco-mask* 32 = (0, 0, 0, 0) <proof>

end

45 Parser for iptables-save

```

theory Parser6
imports Code-Interface
keywords parse-ip6tables-save :: thy-decl
begin

```

<ML>

46 An SML Parser for iptables-save

Work in Progress

<ML>

```

end
theory No-Spoof-Embeddings
imports Semantics-Embeddings
        Primitive-Matchers/No-Spoof
begin

```

47 Spoofing protection in Ternary Semantics implies Spoofing protection Boolean Semantics

If *no-spoofing* is shown in the ternary semantics, it implies that no spoofing is possible in the Boolean semantics with magic oracle. We only assume that the oracle agrees with the *common-matcher* on the not-unknown parts.

lemma *approximating-imp-boolean-semantics-nospoofing*:

assumes *matcher-agree-on-exact-matches* γ *common-matcher*

and *simple-ruleset* rs

and *no-spoofing*: *no-spoofing* *TYPE*('pkt-ext) *ipassmt* rs

shows $\forall iface \in dom\ ipassmt. \forall p :: ('i::len, 'pkt-ext)\ tagged\ packet\ scheme.$

$(\Gamma, \gamma, p \langle p\text{-iface} := iface\text{-sel}\ iface \rangle) \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalAllow)$

→

$p\text{-src}\ p \in (ipcidr\text{-union}\text{-set}\ (set\ (the\ (ipassmt\ iface))))$

$\langle proof \rangle$

corollary

assumes *matcher-agree-on-exact-matches* γ *common-matcher* **and** *simple-ruleset* rs

and *no-spoofing*: *no-spoofing* *TYPE*('pkt-ext) *ipassmt* rs **and** $iface \in dom\ ipassmt$

shows $\{p\text{-src}\ p \mid p :: ('i::len, 'pkt-ext)\ tagged\ packet\ scheme. (\Gamma, \gamma, p \langle p\text{-iface} := iface\text{-sel}\ iface \rangle) \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalAllow)\} \subseteq$

$ipcidr\text{-union}\text{-set}\ (set\ (the\ (ipassmt\ iface)))$

$\langle proof \rangle$

corollary *no-spoofing-executable-set*:

assumes *matcher-agree-on-exact-matches* γ *common-matcher*

and *simple-ruleset* rs

and $\forall r \in set\ rs. normalized\ nnf\ match\ (get\ match\ r)$

and *no-spoofing-executable*: $\forall iface \in dom\ ipassmt. no\ spoofing\ iface\ iface$ *ipassmt* rs

and $iface \in dom\ ipassmt$

shows $\{p\text{-src}\ p \mid p :: ('i::len, 'pkt-ext)\ tagged\ packet\ scheme. (\Gamma, \gamma, p \langle p\text{-iface} := iface\text{-sel}\ iface \rangle) \vdash \langle rs, Undecided \rangle \Rightarrow Decision\ FinalAllow)\} \subseteq$

$ipcidr\text{-union}\text{-set}\ (set\ (the\ (ipassmt\ iface)))$

$\langle proof \rangle$

corollary *no-spoofing-executable-set-preprocessed*:

fixes *ipassmt* $:: 'i::len\ ipassignment$

defines *preprocess* $rs \equiv upper\ closure\ (packet\ assume\ new\ rs)$

and *newpkt* $p \equiv match\ tcp\ flags\ ipt\ tcp\ syn\ (p\ tcp\ flags\ p) \wedge p\ tag\ ctstate$ $p = CT\ New$

assumes *matcher-agree-on-exact-matches* γ *common-matcher*

```

    and simplers: simple-ruleset rs
    and no-spoofing-executable:  $\forall$  iface  $\in$  dom ipassmt. no-spoofing-iface iface
ipassmt (preprocess rs)
    and iface  $\in$  dom ipassmt
    shows {p-src p | p :: ('i::len,'pkt-ext) tagged-packet-scheme. newpkt p  $\wedge$ 
 $\Gamma, \gamma, p \langle p\text{-iface} := \text{iface-sel } \text{iface} \rangle \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Decision FinalAllow} \} \subseteq$ 
        ipcidr-union-set (set (the (ipassmt iface)))
    <proof>

end

```

48 Parser for iptables-save

```

theory Parser
imports Code-Interface
    keywords parse-iptables-save :: thy-decl
begin

```

<ML>

49 An SML Parser for iptables-save

Work in Progress

<ML>

```

end
theory Code-haskell
imports
    Routing.IpRoute-Parser
    Primitive-Matchers/Parser
begin

```

definition *word-less-eq* :: ('a::len) word \Rightarrow ('a::len) word \Rightarrow bool **where**
word-less-eq a b \equiv a \leq b

definition *word-to-nat* :: ('a::len) word \Rightarrow nat **where**
word-to-nat = Word.unat

definition *mk-Set* :: 'a list \Rightarrow 'a set **where**
mk-Set = set

Assumes that you call *fill-l4-protocol* after parsing!

definition *mk-L4Ports-pre* :: raw-ports \Rightarrow ipt-l4-ports **where**
mk-L4Ports-pre ports-raw = L4Ports 0 ports-raw

```
fun ipassmt-iprange-translate :: 'i::len ipt-iprange list negation-type => ('i word ×
nat) list where
  ipassmt-iprange-translate (Pos ips) = concat (map ipt-iprange-to-cidr ips) |
  ipassmt-iprange-translate (Neg ips) = all-but-those-ips (concat (map ipt-iprange-to-cidr
ips))
```

definition *to-ipassmt*

```
:: (iface × 'i::len ipt-iprange list negation-type) list => (iface × ('i word × nat)
list) list where
  to-ipassmt assmt = map (λ(ifce, ips). (ifce, ipassmt-iprange-translate ips)) assmt
```

definition *zero-word* ≡ 0 :: ('a :: len) word

export-code *Rule*

```
Match MatchNot MatchAnd MatchAny
Src Dst Iiface Oiface Prot Src-Ports Dst-Ports CT-State Extra
mk-L4Ports-pre
ProtoAny Proto TCP UDP ICMP L4-Protocol.IPv6ICMP L4-Protocol.SCTP
L4-Protocol.GRE
L4-Protocol.ESP L4-Protocol.AH
Iiface
integer-to-16word nat-to-16word nat-of-integer integer-of-nat word-less-eq word-to-nat
```

nat-to-8word

```
IpAddrNetmask IpAddrRange IpAddr
CT-New CT-Established CT-Related CT-Untracked CT-Invalid
TCP-Flags TCP-SYN TCP-ACK TCP-FIN TCP-RST TCP-URG TCP-PSH
Accept Drop Log Reject Call Return Goto Empty Unknown
action-toString
```

ipv4addr-of-dotdecimal

```
ipt-ipv4range-toString
common-primitive-ipv4-toString
common-primitive-match-expr-ipv4-toString
simple-rule-ipv4-toString
```

mk-ipv6addr IPv6AddrPreferred ipv6preferred-to-int int-to-ipv6preferred

```
ipt-ipv6range-toString
common-primitive-ipv6-toString
common-primitive-match-expr-ipv6-toString
simple-rule-ipv6-toString
```

Semantics-Goto.rewrite-Goto-safe

```
alist-and' compress-parsed-extra fill-l4-protocol Pos Neg mk-Set
unfold-ruleset-CHAIN-safe map-of-string
upper-closure
abstract-for-simple-firewall optimize-matches
```



```

packet-assume-new
to-simple-firewall
to-simple-firewall-without-interfaces
sanity-wf-ruleset
has-default-policy
  ipassmt-generic-ipv4 ipassmt-generic-ipv6
no-spoofing-iface ipassmt-sanity-defined map-of-ipassmt to-ipassmt ipassmt-diff
Pos Neg

simple-fw-valid
debug-ipassmt-ipv4 debug-ipassmt-ipv6

access-matrix-pretty-ipv4 access-matrix-pretty-ipv6
mk-parts-connection-TCP

PrefixMatch routing-rule-ext routing-action-ext
routing-action-oiface-update metric-update routing-action-next-hop-update empty-rr-hlp
sort-rtbl
  prefix-match-32-toString routing-rule-32-toString prefix-match-128-toString rout-
ing-rule-128-toString
  default-prefix sanity-ip-route ipassmt-diff routing-ipassmt
  checking SML Haskell?

end
theory Access-Matrix-Embeddings
imports Semantics-Embeddings
  Primitive-Matchers/No-Spoof
  Simple-Firewall.Service-Matrix
begin

```

50 Applying the Access Matrix to the Bigstep Semantics

If the real iptables firewall (*iptables-bigstep*) accepts a packet, we have a corresponding edge in the *access-matrix*.

corollary *access-matrix-and-bigstep-semantics*:

```

defines preprocess rs  $\equiv$  upper-closure (optimize-matches abstract-for-simple-firewall
(upper-closure (packet-assume-new rs)))
and newpkt p  $\equiv$  match-tcp-flags ipt-tcp-syn (p-tcp-flags p)  $\wedge$  p-tag-ctstate p
= CT-New
fixes  $\gamma :: 'i::len$  common-primitive  $\Rightarrow ('i, 'pkt-ext)$  tagged-packet-scheme  $\Rightarrow$  bool
and p :: ('i::len, 'pkt-ext) tagged-packet-scheme
assumes agree:matcher-agree-on-exact-matches  $\gamma$  common-matcher
and simple: simple-ruleset rs
and new: newpkt p
and matrix:  $(V, E) =$  access-matrix ( $\downarrow$ pc-iiface = p-iiface p, pc-oiface =
p-oiface p, pc-proto = p-proto p, pc-sport = p-sport p, pc-dport = p-dport p)

```

(*to-simple-firewall* (*preprocess* *rs*))
and *accept*: $\Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Decision FinalAllow}$
shows $\exists s\text{-repr } d\text{-repr } s\text{-range } d\text{-range}. (s\text{-repr}, d\text{-repr}) \in \text{set } E \wedge$
 $(\text{map-of } V) s\text{-repr} = \text{Some } s\text{-range} \wedge (p\text{-src } p) \in \text{wordinterval-to-set}$
 $s\text{-range} \wedge$
 $(\text{map-of } V) d\text{-repr} = \text{Some } d\text{-range} \wedge (p\text{-dst } p) \in \text{wordinterval-to-set}$
 $d\text{-range}$
 $\langle \text{proof} \rangle$

corollary *access-matrix-no-interfaces-and-bigstep-semantics*:

defines *newpkt* $p \equiv \text{match-tcp-flags } \text{ipt-tcp-syn } (p\text{-tcp-flags } p) \wedge p\text{-tag-ctstate } p$
 $= \text{CT-New}$

fixes $\gamma :: 'i::\text{len } \text{common-primitive} \Rightarrow ('i, 'pkt\text{-ext}) \text{tagged-packet-scheme} \Rightarrow \text{bool}$

and $p :: ('i::\text{len}, 'pkt\text{-ext}) \text{tagged-packet-scheme}$

assumes *agree:matcher-agree-on-exact-matches* γ *common-matcher*

and *simple: simple-ruleset* *rs*

— To get the best results, we want to rewrite all interfaces, which needs some preconditions

— well-formed *ipassmt*

and *wf-ipassmt1*: *ipassmt-sanity-nowildcards* (*map-of ipassmt*) **and** *wf-ipassmt2*:
distinct (*map fst ipassmt*)

— There are no spoofed packets (probably by kernel's reverse path filter or our checker). This assumption implies that *ipassmt* lists ALL interfaces (!!).

and *nospoofing*: $\forall (p::('i::\text{len}, 'pkt\text{-ext}) \text{tagged-packet-scheme}).$

$\exists \text{ips}. (\text{map-of } \text{ipassmt}) (\text{Iface } (p\text{-iface } p)) = \text{Some } \text{ips} \wedge p\text{-src } p \in$
 $\text{ipcidr-union-set } (\text{set } \text{ips})$

— If a routing table was passed, the output interface for any packet we consider is decided based on it.

and *routing-decided*: $\bigwedge \text{rtbl } (p::('i, 'pkt\text{-ext}) \text{tagged-packet-scheme}). \text{rtblo} =$
 $\text{Some } \text{rtbl} \implies \text{output-iface } (\text{routing-table-semantics } \text{rtbl } (p\text{-dst } p)) = p\text{-iface } p$

— A passed routing table is wellformed

and *correct-routing*: $\bigwedge \text{rtbl}. \text{rtblo} = \text{Some } \text{rtbl} \implies \text{correct-routing } \text{rtbl}$

— A passed routing table contains no interfaces with wildcard names

and *routing-no-wildcards*: $\bigwedge \text{rtbl}. \text{rtblo} = \text{Some } \text{rtbl} \implies \text{ipassmt-sanity-nowildcards}$
 $(\text{map-of } (\text{routing-ipassmt } \text{rtbl}))$

and *new*: *newpkt* p

— building the matrix over ANY interfaces, not mentioned anywhere. That means, we don't care about interfaces!

and *matrix*: $(V, E) = \text{access-matrix } (\text{pc-iface} = \text{anyI}, \text{pc-oiface} = \text{anyO},$
 $\text{pc-proto} = p\text{-proto } p, \text{pc-sport} = p\text{-sport } p, \text{pc-dport} = p\text{-dport } p)$

(*to-simple-firewall-without-interfaces ipassmt* *rtblo* *rs*)

and *accept*: $\Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Decision FinalAllow}$

shows $\exists s\text{-repr } d\text{-repr } s\text{-range } d\text{-range}. (s\text{-repr}, d\text{-repr}) \in \text{set } E \wedge$

$(\text{map-of } V) s\text{-repr} = \text{Some } s\text{-range} \wedge (p\text{-src } p) \in \text{wordinterval-to-set}$
 $s\text{-range} \wedge$

$(\text{map-of } V) d\text{-repr} = \text{Some } d\text{-range} \wedge (p\text{-dst } p) \in \text{wordinterval-to-set}$
 $d\text{-range}$

<proof>

```
end  
theory Documentation  
imports Semantics-Embeddings  
         Call-Return-Unfolding  
         No-Spoof-Embeddings  
         Access-Matrix-Embeddings  
         Primitive-Matchers/Code-Interface  
begin
```

51 Documentation

51.1 General Model

The semantics of the filtering behavior of iptables is expressed by *iptables-bigstep*. The notation $\Gamma, \gamma, p \vdash_g \langle rs, s \rangle \Rightarrow t$ reads as follows: Γ is the background ruleset (user-defined rules). γ is a function (*'a, 'p*) *matcher* which is called the primitive matcher (i.e. the matching features supported by iptables). p is the packet inspected by the firewall. rs is the ruleset. s and t are the start state and final state.

The semantics:

$$\frac{\Gamma, \gamma, p \vdash_g \langle [], t \rangle \Rightarrow t}{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ action.Accept], Undecided \rangle \Rightarrow Decision\ FinalAllow}$$
$$\frac{Semantics-Goto.matches\ \gamma\ m\ p}{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ action.Drop], Undecided \rangle \Rightarrow Decision\ FinalDeny}$$
$$\frac{Semantics-Goto.matches\ \gamma\ m\ p}{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ Reject], Undecided \rangle \Rightarrow Decision\ FinalDeny}$$
$$\frac{Semantics-Goto.matches\ \gamma\ m\ p}{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ Log], Undecided \rangle \Rightarrow Undecided}$$
$$\frac{Semantics-Goto.matches\ \gamma\ m\ p}{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ Empty], Undecided \rangle \Rightarrow Undecided}$$
$$\frac{\neg Semantics-Goto.matches\ \gamma\ m\ p}{\Gamma, \gamma, p \vdash_g \langle [Rule\ m\ a], Undecided \rangle \Rightarrow Undecided}$$
$$\Gamma, \gamma, p \vdash_g \langle rs, Decision\ X \rangle \Rightarrow Decision\ X$$

$$\begin{array}{c}
\frac{\Gamma, \gamma, p \vdash_g \langle rs_1, \text{Undecided} \rangle \Rightarrow t \quad \Gamma, \gamma, p \vdash_g \langle rs_2, t \rangle \Rightarrow t' \quad \text{Semantics-Goto.no-matching-Goto } \gamma \ p \ rs_1}{\Gamma, \gamma, p \vdash_g \langle rs_1 @ rs_2, \text{Undecided} \rangle \Rightarrow t'} \\
\frac{\text{Semantics-Goto.matches } \gamma \ m \ p \quad \Gamma \ \text{chain} = \text{Some } (rs_1 @ [\text{Rule } m' \ \text{Return}] @ rs_2) \quad \text{Semantics-Goto.matches } \gamma \ m' \ p \quad \Gamma, \gamma, p \vdash_g \langle rs_1, \text{Undecided} \rangle \Rightarrow \text{Undecided} \quad \text{Semantics-Goto.no-matching-Goto } \gamma \ p \ rs_1}{\Gamma, \gamma, p \vdash_g \langle [\text{Rule } m \ (\text{Call chain})], \text{Undecided} \rangle \Rightarrow \text{Undecided}} \\
\frac{\text{Semantics-Goto.matches } \gamma \ m \ p \quad \Gamma \ \text{chain} = \text{Some } rs \quad \Gamma, \gamma, p \vdash_g \langle rs, \text{Undecided} \rangle \Rightarrow t}{\Gamma, \gamma, p \vdash_g \langle [\text{Rule } m \ (\text{Call chain})], \text{Undecided} \rangle \Rightarrow t}
\end{array}$$

51.2 Unfolding the Ruleset

We can replace all *Gotos* to terminal chains (chains that ultimately yield a final decision for every packet) with *Calls*. Otherwise we don't have as rich goto semantics as iptables has, but this rewriting is safe.

Semantics-Goto.rewrite-Goto-chain-safe $\Gamma \ rs = \text{Some } rs' \implies \Gamma, \gamma, p \vdash_g \langle rs', s \rangle \Rightarrow t = \Gamma, \gamma, p \vdash_g \langle rs, s \rangle \Rightarrow t$

The iptables firewall starts as follows: [*Rule MatchAny (Call chain-name), Rule MatchAny default-action*] We call to a built-in chain *chain-name*, usually INPUT, OUTPUT, or FORWARD. If we don't get a decision, iptables uses the default policy (-P) *default-action*.

We can call *unfold-optimize-ruleset-CHAIN* to remove all calls to user-defined chains and other unpleasant actions. We get back a *simple-ruleset* which has exactly the same behaviour. As a bonus, this *simple-ruleset* already has some match conditions optimized.

For example, if the parser does not find a source IP in a rule, it is okay to specify -s 0.0.0.0/0, the unfolding will optimize away these things for you. Or if you parse iptables -L -n which always has these annoying 0.0.0.0/0 fields. May make the parser easier. The following lemma shows that this does not change the semantics.

lemma *unfold-optimize-common-matcher-univ-ruleset-CHAIN*:

— for IPv4 and IPv6 packets

fixes $\gamma :: 'i::len \ \text{common-primitive} \Rightarrow ('i, 'pkt\text{-ext}) \ \text{tagged-packet-scheme} \Rightarrow \text{bool}$

and $p :: ('i::len, 'pkt\text{-ext}) \ \text{tagged-packet-scheme}$

assumes *sanity-wf-ruleset* Γ **and** $\text{chain-name} \in \text{set } (\text{map } \text{fst } \Gamma)$

and $\text{default-action} = \text{action.Accept} \vee \text{default-action} = \text{action.Drop}$

and *matcher-agree-on-exact-matches* $\gamma \ \text{common-matcher}$

and *unfold-ruleset-CHAIN-safe* $\text{chain-name } \text{default-action} (\text{map-of } \Gamma) = \text{Some}$

rs

shows $(\text{map-of } \Gamma), \gamma, p \vdash \langle rs, s \rangle \Rightarrow t \iff$
 $(\text{map-of } \Gamma), \gamma, p \vdash \langle [\text{Rule MatchAny } (\text{Call chain-name}), \text{Rule MatchAny}$
default-action], $s \rangle \Rightarrow t$
and *simple-ruleset* rs
 $\langle \text{proof} \rangle$

51.3 Spoofing protection

We provide an executable algorithm *no-spoofing-iface* which checks that a ruleset provides spoofing protection:

$\llbracket \text{matcher-agree-on-exact-matches } \gamma \text{ common-matcher; simple-ruleset } rs; \forall r \in \text{set}$
 $rs. \text{normalized-nmf-match } (\text{get-match } r); \forall \text{iface} \in \text{dom } ipassmt. \text{no-spoofing-iface}$
 $\text{iface } ipassmt \text{ } rs; \text{iface} \in \text{dom } ipassmt \rrbracket \implies \{p\text{-src } p \mid \Gamma, \gamma, p \vdash (p\text{-iface} :=$
 $\text{iface-sel } \text{iface}) \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Decision FinalAllow}\} \subseteq \text{ipcidr-union-set}$
 $(\text{set } (\text{the } (ipassmt \text{ iface})))$

Text the firewall needs normalized match conditions, this is a good way to preprocess the firewall before checking spoofing protection:

$\llbracket \text{matcher-agree-on-exact-matches } \gamma \text{ common-matcher; simple-ruleset } rs; \forall \text{iface} \in \text{dom}$
 $ipassmt. \text{no-spoofing-iface } \text{iface } ipassmt (\text{upper-closure } (\text{packet-assume-new}$
 $rs)); \text{iface} \in \text{dom } ipassmt \rrbracket \implies \{p\text{-src } p \mid (\text{match-tcp-flags } \text{ipt-tcp-syn } (p\text{-tcp-flags}$
 $p) \wedge p\text{-tag-ctstate } p = \text{CT-New}) \wedge \Gamma, \gamma, p \vdash (p\text{-iface} := \text{iface-sel } \text{iface}) \vdash \langle rs,$
 $\text{Undecided} \rangle \Rightarrow \text{Decision FinalAllow}\} \subseteq \text{ipcidr-union-set } (\text{set } (\text{the } (ipassmt$
 $\text{iface})))$

51.4 Simple Firewall Model

The simple firewall supports the following match conditions: *i* *simple-match*.

The *simple-fw* model is remarkably simple: *simple-fw* $\llbracket uu = \text{Undecided}$
simple-fw (*SimpleRule* m *simple-action*.*Accept* $\cdot rs$) $p = (\text{if } \text{simple-matches}$
 $m \text{ } p \text{ then } \text{Decision FinalAllow} \text{ else } \text{simple-fw } rs \text{ } p)$

simple-fw (*SimpleRule* m *simple-action*.*Drop* $\cdot rs$) $p = (\text{if } \text{simple-matches } m$
 $p \text{ then } \text{Decision FinalDeny} \text{ else } \text{simple-fw } rs \text{ } p)$

We support translating to a stricter version (a version that accepts less packets):

$\llbracket \text{matcher-agree-on-exact-matches } \gamma \text{ common-matcher; simple-ruleset } rs \rrbracket \implies$
 $\{p \mid \text{simple-fw } (\text{to-simple-firewall } (\text{lower-closure } (\text{optimize-matches } \text{abstract-for-simple-firewall}$
 $(\text{lower-closure } (\text{packet-assume-new } rs)))))) \text{ } p = \text{Decision FinalAllow} \wedge \text{match-tcp-flags}$
 $\text{ipt-tcp-syn } (p\text{-tcp-flags } p) \wedge p\text{-tag-ctstate } p = \text{CT-New}\} \subseteq \{p \mid \Gamma, \gamma, p \vdash \langle rs,$
 $\text{Undecided} \rangle \Rightarrow \text{Decision FinalAllow} \wedge \text{match-tcp-flags } \text{ipt-tcp-syn } (p\text{-tcp-flags}$
 $p) \wedge p\text{-tag-ctstate } p = \text{CT-New}\}$

We support translating to a more permissive version (a version that accepts more packets):

$\llbracket \text{matcher-agree-on-exact-matches } \gamma \text{ common-matcher; simple-ruleset } rs \rrbracket \implies$

$$\{p \mid \Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Decision FinalAllow} \wedge \text{match-tcp-flags ipt-tcp-syn} \\ (p\text{-tcp-flags } p) \wedge p\text{-tag-ctstate } p = \text{CT-New}\} \subseteq \{p \mid \text{simple-fw (to-simple-firewall} \\ (\text{upper-closure (optimize-matches abstract-for-simple-firewall (upper-closure \\ (\text{packet-assume-new } rs)))))) p = \text{Decision FinalAllow} \wedge \text{match-tcp-flags ipt-tcp-syn} \\ (p\text{-tcp-flags } p) \wedge p\text{-tag-ctstate } p = \text{CT-New}\}$$

There is also a different approach to translate to the simple firewall which removes all matches on interfaces:

$$\llbracket \text{simple-ruleset } rs; \text{ipassmt-sanity-nowildcards (map-of ipassmt); distinct (map} \\ \text{fst ipassmt); } \forall p. \exists ips. \text{map-of ipassmt (Iface (p-iiface } p)) = \text{Some ips} \wedge \\ p\text{-src } p \in \text{ipcidr-union-set (set ips); } \bigwedge \text{rtbl } p. \text{rtblo} = \text{Some rtbl} \Rightarrow \text{out-} \\ \text{put-iface (routing-table-semantics rtbl (p-dst } p)) = p\text{-oiface } p; \bigwedge \text{rtbl. rt-} \\ \text{blo} = \text{Some rtbl} \Rightarrow \text{correct-routing rtbl; } \bigwedge \text{rtbl. rtblo} = \text{Some rtbl} \Rightarrow \\ \text{ipassmt-sanity-nowildcards (map-of (routing-ipassmt rtbl))} \rrbracket \Rightarrow \{p \mid (\text{common-matcher,} \\ \text{in-doubt-allow}), p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow_{\alpha} \text{Decision FinalAllow} \wedge \text{match-tcp-flags} \\ \text{ipt-tcp-syn (p-tcp-flags } p) \wedge p\text{-tag-ctstate } p = \text{CT-New}\} \subseteq \{p \mid \text{simple-fw} \\ (\text{to-simple-firewall-without-interfaces ipassmt rtblo } rs) p = \text{Decision FinalAl-} \\ \text{low} \wedge \text{match-tcp-flags ipt-tcp-syn (p-tcp-flags } p) \wedge p\text{-tag-ctstate } p = \text{CT-New}\}$$

$$\llbracket \text{simple-ruleset } rs; \text{ipassmt-sanity-nowildcards (map-of ipassmt); distinct (map} \\ \text{fst ipassmt); } \forall p. \exists ips. \text{map-of ipassmt (Iface (p-iiface } p)) = \text{Some ips} \wedge \\ p\text{-src } p \in \text{ipcidr-union-set (set ips); } \bigwedge \text{rtbl } p. \text{rtblo} = \text{Some rtbl} \Rightarrow \text{out-} \\ \text{put-iface (routing-table-semantics rtbl (p-dst } p)) = p\text{-oiface } p; \bigwedge \text{rtbl. rt-} \\ \text{blo} = \text{Some rtbl} \Rightarrow \text{correct-routing rtbl; } \bigwedge \text{rtbl. rtblo} = \text{Some rtbl} \Rightarrow \\ \text{ipassmt-sanity-nowildcards (map-of (routing-ipassmt rtbl))} \rrbracket \Rightarrow \forall r \in \text{set (to-simple-firewall-without-ir} \\ \text{ipassmt rtblo } rs). \text{iiface (match-sel } r) = \text{ifaceAny} \wedge \text{oiface (match-sel } r) = \\ \text{ifaceAny}$$

51.5 Service Matrices

For a *'i* simple-rule list and a fixed *parts-connection*, we support to partition the IPv4 address space the following.

All members of a partition have the same access rights: $V \in \text{set (build-ip-partition } c \text{ } rs) \Rightarrow \forall ip1 \in \text{wordinterval-to-set } V. \forall ip2 \in \text{wordinterval-to-set } V. \text{same-fw-behaviour-one } ip1 \text{ } ip2 \text{ } c \text{ } rs$

Minimal: $\llbracket A \in \text{set (build-ip-partition } c \text{ } rs); B \in \text{set (build-ip-partition } c \text{ } rs); A \neq B \rrbracket \Rightarrow \forall ip1 \in \text{wordinterval-to-set } A. \forall ip2 \in \text{wordinterval-to-set } B. \neg \text{same-fw-behaviour-one } ip1 \text{ } ip2 \text{ } c \text{ } rs$

The resulting access control matrix is sound and complete:

$$(V, E) = \text{access-matrix } c \text{ } rs \Rightarrow (\exists s\text{-repr } d\text{-repr } s\text{-range } d\text{-range. (s-repr,} \\ d\text{-repr)} \in \text{set } E \wedge \text{map-of } V \text{ } s\text{-repr} = \text{Some } s\text{-range} \wedge s \in \text{wordinter-} \\ \text{val-to-set } s\text{-range} \wedge \text{map-of } V \text{ } d\text{-repr} = \text{Some } d\text{-range} \wedge d \in \text{wordinter-} \\ \text{val-to-set } d\text{-range}) = (\text{runFw } s \text{ } d \text{ } c \text{ } rs = \text{Decision FinalAllow})$$

Theorem reads: For a fixed connection, you can look up IP addresses (source

and destination pairs) in the matrix if and only if the firewall accepts this src,dst IP address pair for the fixed connection. Note: The matrix is actually a graph (nice visualization!), you need to look up IP addresses in the Vertices and check the access of the representants in the edges. If you want to visualize the graph (e.g. with Graphviz or tkiz): The vertices are the node description (i.e. header; *dom V* is the label for each node which will also be referenced in the edges, *ran V* is the human-readable description for each node (i.e. the full IP range it represents)), the edges are the edges. Result looks nice. Theorem also tells us that this visualization is correct.

A final theorem which does not mention the simple firewall at all. If the real iptables firewall (*iptables-bigstep*) accepts a packet, we have a corresponding edge in the *access-matrix*:

$$\begin{aligned} & \llbracket \text{matcher-agree-on-exact-matches } \gamma \text{ common-matcher; simple-ruleset } rs; \text{match-tcp-flags} \\ & \text{ipt-tcp-syn } (p\text{-tcp-flags } p) \wedge p\text{-tag-ctstate } p = CT\text{-New; } (V, E) = \text{access-matrix} \\ & (\!|pc\text{-iface} = p\text{-iface } p, pc\text{-oiface} = p\text{-oiface } p, pc\text{-proto} = p\text{-proto } p, pc\text{-sport} \\ & = p\text{-sport } p, pc\text{-dport} = p\text{-dport } p) \text{ (to-simple-firewall (upper-closure (optimize-matches} \\ & \text{abstract-for-simple-firewall (upper-closure (packet-assume-new } rs))))); \Gamma, \gamma, p \vdash \\ & \langle rs, \text{Undecided} \rangle \Rightarrow \text{Decision FinalAllow} \rrbracket \Longrightarrow \exists s\text{-repr } d\text{-repr } s\text{-range } d\text{-range.} \\ & (s\text{-repr}, d\text{-repr}) \in \text{set } E \wedge \text{map-of } V \text{ } s\text{-repr} = \text{Some } s\text{-range} \wedge p\text{-src } p \in \\ & \text{wordinterval-to-set } s\text{-range} \wedge \text{map-of } V \text{ } d\text{-repr} = \text{Some } d\text{-range} \wedge p\text{-dst } p \in \\ & \text{wordinterval-to-set } d\text{-range} \end{aligned}$$

Actually, we want to ignore all interfaces for a service matrix. This is done in $\llbracket \text{matcher-agree-on-exact-matches } \gamma \text{ common-matcher; simple-ruleset } rs; \text{ipassmt-sanity-nowildcards (map-of ipassmt); distinct (map fst ipassmt); } \forall p. \exists ips. \text{map-of ipassmt (Iface (p-iface } p)) = \text{Some } ips \wedge p\text{-src } p \in \text{ipcidr-union-set (set ips); } \bigwedge \text{rtbl } p. \text{rtblo} = \text{Some } \text{rtbl} \Longrightarrow \text{output-iface (routing-table-semantics rtbl (p-dst } p)) = p\text{-oiface } p; \bigwedge \text{rtbl. } \text{rtblo} = \text{Some } \text{rtbl} \Longrightarrow \text{correct-routing rtbl; } \bigwedge \text{rtbl. } \text{rtblo} = \text{Some } \text{rtbl} \Longrightarrow \text{ipassmt-sanity-nowildcards (map-of (routing-ipassmt rtbl)); match-tcp-flags ipt-tcp-syn } (p\text{-tcp-flags } p) \wedge p\text{-tag-ctstate } p = CT\text{-New; } (V, E) = \text{access-matrix} (\!|pc\text{-iface} = \text{anyI}, pc\text{-oiface} = \text{anyO}, pc\text{-proto} = p\text{-proto } p, pc\text{-sport} = p\text{-sport } p, pc\text{-dport} = p\text{-dport } p) \text{ (to-simple-firewall-without-interfaces ipassmt rtblo } rs); \Gamma, \gamma, p \vdash \langle rs, \text{Undecided} \rangle \Rightarrow \text{Decision FinalAllow} \rrbracket \Longrightarrow \exists s\text{-repr } d\text{-repr } s\text{-range } d\text{-range. } (s\text{-repr}, d\text{-repr}) \in \text{set } E \wedge \text{map-of } V \text{ } s\text{-repr} = \text{Some } s\text{-range} \wedge p\text{-src } p \in \text{wordinterval-to-set } s\text{-range} \wedge \text{map-of } V \text{ } d\text{-repr} = \text{Some } d\text{-range} \wedge p\text{-dst } p \in \text{wordinterval-to-set } d\text{-range.}$ The theorem reads a bit ugly because we need well-formedness assumptions if we rewrite interfaces. Internally, it uses *iface-try-rewrite* which is pretty safe to use, even if you don't have an *ipassmt* or routing tables.

end

References

- [1] C. Diekmann. Verified Firewall Ruleset Verification – Math, Functional Programming, Theorem Proving, and an Introduction to Isabelle/HOL. 32. Chaos Communication Congress (32C3), 12 2015. Recording: https://media.ccc.de/v/32c3-7195-verified_firewall_ruleset_verification.
- [2] C. Diekmann, L. Hupel, and G. Carle. Semantics-Preserving Simplification of Real-World Firewall Rule Sets. In *20th International Symposium on Formal Methods*, pages 195–212. Springer, 6 2015. http://www.net.in.tum.de/fileadmin/bibtex/publications/papers/fm15_Semantics-Preserving_Simplification_of_Real-World_Firewall_Rule_Sets.pdf.
- [3] C. Diekmann, J. Michaelis, M. Haslbeck, and G. Carle. Verified iptables Firewall Analysis. In *IFIP Networking 2016*, Vienna, Austria, 5 2016. <http://dl.ifip.org/db/conf/networking/networking2016/1570232858.pdf>.
- [4] C. Diekmann, L. Schwaighofer, and G. Carle. Certifying Spoofing-Protection of Firewalls. In *11th International Conference on Network and Service Management, CNSM*, Barcelona, Spain, 11 2015. http://www.net.in.tum.de/fileadmin/bibtex/publications/papers/diekmann2015_cnsm.pdf.