Iptables-Semantics

Cornelius Diekmann, Lars Hupel

May 27, 2015

Contents

theory *Firewall-Common-Decision-State*
imports *Main*
begin

datatype *final-decision* = *FinalAllow* | *FinalDeny*

The state during packet processing. If undecided, there are some remaining rules to process. If decided, there is an action which applies to the packet

datatype *state* = *Undecided* | *Decision final-decision*

end
theory *Firewall-Common*
imports *Main Firewall-Common-Decision-State*
begin

1 Firewall Basic Syntax

Our firewall model supports the following actions.

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

The type parameter $'a$ denotes the primitive match condition For example, matching on source IP address or on protocol. We list 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*

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

datatype-compat *rule*

end
theory *Misc*
imports *Main*
begin

lemma *list-app-singletonE*:
  assumes rs_1 @ rs_2 = [x]
  obtains (*first*) rs_1 = [x] rs_2 = []
    | (*second*) rs_1 = [] rs_2 = [x]
using *assms*
by (*cases* rs_1) *auto*

lemma *list-app-eq-cases*:
  assumes xs_1 @ xs_2 = ys_1 @ ys_2
  obtains (*longer*) xs_1 = *take* (*length* xs_1) ys_1 xs_2 = *drop* (*length* xs_1) ys_1 @ ys_2
    | (*shorter*) ys_1 = *take* (*length* ys_1) xs_1 ys_2 = *drop* (*length* ys_1) xs_1 @ xs_2
using *assms*
apply (*cases length* xs_1 ≤ *length* ys_1)
apply (*metis append-eq-append-conv-if*)+
done

end
theory *Semantics*
imports *Main Firewall-Common Misc ~~/src/HOL/Library/LaTeXsugar*
begin

2 Big Step Semantics

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

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

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

fun *matches* :: $('a,\ 'p)$ *matcher* $\Rightarrow\ 'a$ *match-expr* $\Rightarrow\ 'p \Rightarrow$ *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*) $p \longleftrightarrow \gamma\ e\ p$ |
matches - *MatchAny* - $\longleftrightarrow$ *True*

inductive *iptables-bigstep* :: $'a$ *ruleset* $\Rightarrow ('a,\ 'p)$ *matcher* $\Rightarrow\ 'p \Rightarrow\ 'a$ *rule list* $\Rightarrow$ *state* $\Rightarrow$ *state* $\Rightarrow$ *bool*
(-,-,-$\vdash$ $\langle$-, -$\rangle$ $\Rightarrow$ - [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* $\gamma\ m\ p \Longrightarrow \Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Accept}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalAllow}$ |
drop: *matches* $\gamma\ m\ p \Longrightarrow \Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Drop}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalDeny}$ |
reject: *matches* $\gamma\ m\ p \Longrightarrow \Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Reject}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalDeny}$ |
log: *matches* $\gamma\ m\ p \Longrightarrow \Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Log}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}$ |

empty: *matches* $\gamma\ m\ p \Longrightarrow \Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Empty}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}$ |
nomatch: $\neg$ *matches* $\gamma\ m\ p \Longrightarrow \Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ a],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}$ |
decision: $\Gamma,\gamma,p\vdash \langle rs,\ \mathit{Decision}\ X\rangle \Rightarrow \mathit{Decision}\ X$ |
seq: $[\![\Gamma,\gamma,p\vdash \langle rs_1,\ \mathit{Undecided}\rangle \Rightarrow t;\ \Gamma,\gamma,p\vdash \langle rs_2,\ t\rangle \Rightarrow t']\!] \Longrightarrow \Gamma,\gamma,p\vdash \langle rs_1@rs_2,\ \mathit{Undecided}\rangle \Rightarrow t'$ |
call-return: $[\![$ *matches* $\gamma\ m\ p$; Γ *chain* = *Some* $(rs_1@[\mathit{Rule}\ m'\ \mathit{Return}]@rs_2)$;
matches $\gamma\ m'\ p$; $\Gamma,\gamma,p\vdash \langle rs_1,\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}\ ]\!] \Longrightarrow$
$\Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ (\mathit{Call}\ \mathit{chain})],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}$ |
call-result: $[\![$ *matches* $\gamma\ m\ p$; Γ *chain* = *Some rs*; $\Gamma,\gamma,p\vdash \langle rs,\ \mathit{Undecided}\rangle \Rightarrow t\ ]\!]$ $\Longrightarrow$
$\Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ (\mathit{Call}\ \mathit{chain})],\ \mathit{Undecided}\rangle \Rightarrow t$

The semantic rules again in pretty format:

$$\frac{}{\Gamma,\gamma,p\vdash \langle[],\ t\rangle \Rightarrow t}$$

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

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

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

$$\frac{matches\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash\ \langle[Rule\ m\ Log],\ Undecided\rangle\ \Rightarrow\ Undecided}$$

$$\frac{matches\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash\ \langle[Rule\ m\ Empty],\ Undecided\rangle\ \Rightarrow\ Undecided}$$

$$\frac{\neg\ matches\ \gamma\ m\ p}{\Gamma,\gamma,p\vdash\ \langle[Rule\ m\ a],\ Undecided\rangle\ \Rightarrow\ Undecided}$$

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

$$\frac{\Gamma,\gamma,p\vdash\ \langle rs_1,\ Undecided\rangle\ \Rightarrow\ t \qquad \Gamma,\gamma,p\vdash\ \langle rs_2,\ t\rangle\ \Rightarrow\ t'}{\Gamma,\gamma,p\vdash\ \langle rs_1\ @\ rs_2,\ Undecided\rangle\ \Rightarrow\ t'}$$

$$\frac{\begin{array}{c}matches\ \gamma\ m\ p \qquad \Gamma\ chain = Some\ (rs_1\ @\ [Rule\ m'\ Return]\ @\ rs_2)\\ matches\ \gamma\ m'\ p \qquad \Gamma,\gamma,p\vdash\ \langle rs_1,\ Undecided\rangle\ \Rightarrow\ Undecided\end{array}}{\Gamma,\gamma,p\vdash\ \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle\ \Rightarrow\ Undecided}$$

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

lemma *deny*:
  matches γ *m p* $\Longrightarrow$ *a* = *Drop* $\vee$ *a* = *Reject* $\Longrightarrow$ *iptables-bigstep* Γ γ *p* [*Rule m a*] *Undecided* (*Decision FinalDeny*)
by (*auto intro*: *drop reject*)

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 –
  from *assms* **have** $\Gamma,\gamma,p\vdash\ \langle[r]\ @\ rs,\ Undecided\rangle\ \Rightarrow\ t'$ **by** (*rule seq*)
  thus *?thesis* **by** *simp*
qed

lemma *iptables-bigstep-induct*
  [*case-names Skip Allow Deny Log Nomatch Decision Seq Call-return Call-result*,
  induct pred: *iptables-bigstep*]:
  $\llbracket$ $\Gamma,\gamma,p\vdash\ \langle rs,s\rangle\ \Rightarrow\ t$;
    $\bigwedge t.\ P\ []\ t\ t$;
    $\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Accept \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalAllow)$;
    $\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Drop \vee a = Reject \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalDeny)$;
    $\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Log \vee a = Empty \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided$;
    $\bigwedge m\ a.\ \neg\ matches\ \gamma\ m\ p \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided$;
    $\bigwedge rs\ X.\ P\ rs\ (Decision\ X)\ (Decision\ X)$;
    $\bigwedge rs\ rs_1\ rs_2\ t\ t'.\ rs = rs_1\ @\ rs_2 \Longrightarrow \Gamma,\gamma,p\vdash\ \langle rs_1,Undecided\rangle\ \Rightarrow\ t \Longrightarrow P\ rs_1\ Undecided\ t \Longrightarrow \Gamma,\gamma,p\vdash\ \langle rs_2,t\rangle\ \Rightarrow\ t' \Longrightarrow P\ rs_2\ t\ t' \Longrightarrow P\ rs\ Undecided\ t'$;
    $\bigwedge m\ a\ chain\ rs_1\ m'\ rs_2.\ matches\ \gamma\ m\ p \Longrightarrow a = Call\ chain \Longrightarrow \Gamma\ chain = Some$

$(rs_1 \mathbin{@} [Rule\ m'\ Return] \mathbin{@} rs_2) \Longrightarrow matches\ \gamma\ m'\ p \Longrightarrow \Gamma,\gamma,p\vdash \langle rs_1, Undecided\rangle \Rightarrow Undecided \Longrightarrow P\ rs_1\ Undecided\ Undecided \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided$;
$\bigwedge m\ a\ chain\ rs\ t.\ matches\ \gamma\ m\ p \Longrightarrow a = Call\ chain \Longrightarrow \Gamma\ chain = Some\ rs \Longrightarrow \Gamma,\gamma,p\vdash \langle rs, Undecided\rangle \Rightarrow t \Longrightarrow P\ rs\ Undecided\ t \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ t\ \rrbracket \Longrightarrow$
$P\ rs\ s\ t$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *skipD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [] \Longrightarrow s = t$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *decisionD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow s = Decision\ X \Longrightarrow t = Decision\ X$
by (*induction rule*: *iptables-bigstep-induct*) *auto*

context
notes *skipD*[*dest*] *list-app-singletonE*[*elim*]
begin

lemma *acceptD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Accept] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Decision\ FinalAllow$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *dropD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Drop] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Decision\ FinalDeny$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *rejectD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Reject] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Decision\ FinalDeny$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *logD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Log] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Undecided$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *emptyD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ Empty] \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow t = Undecided$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *nomatchD*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [Rule\ m\ a] \Longrightarrow s = Undecided \Longrightarrow \neg\ matches\ \gamma\ m\ p \Longrightarrow t = Undecided$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

lemma *callD*:
assumes $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t$ $r = [Rule\ m\ (Call\ chain)]$ $s = Undecided$ $matches\ \gamma\ m\ p$ $\Gamma\ chain = Some\ rs$
obtains $\Gamma,\gamma,p\vdash \langle rs,s\rangle \Rightarrow t$
| $rs_1\ rs_2\ m'$ **where** $rs = rs_1 \mathbin{@} Rule\ m'\ Return\ \#\ rs_2$ $matches\ \gamma\ m'\ p$ $\Gamma,\gamma,p\vdash \langle rs_1,s\rangle \Rightarrow Undecided$ $t = Undecided$
using *assms*

proof (*induction r s t arbitrary*: *rs rule*: *iptables-bigstep.induct*)
  case (*seq* rs_1)
  thus *?case* **by** (*cases* rs_1) *auto*
qed *auto*

end

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

lemma *seq′*:
  assumes $rs = rs_1$ @ rs_2 $\Gamma,\gamma,p\vdash \langle rs_1,s\rangle \Rightarrow t$ $\Gamma,\gamma,p\vdash \langle rs_2,t\rangle \Rightarrow t'$
  shows $\Gamma,\gamma,p\vdash \langle rs,s\rangle \Rightarrow t'$
using *assms* **by** (*cases s*) (*auto intro*: *seq decision dest*: *decisionD*)

lemma *seq′-cons*: $\Gamma,\gamma,p\vdash \langle [r],s\rangle \Rightarrow t \Longrightarrow \Gamma,\gamma,p\vdash \langle rs,t\rangle \Rightarrow t' \Longrightarrow \Gamma,\gamma,p\vdash \langle r\#rs, s\rangle \Rightarrow t'$
by (*metis decision decisionD state.exhaust seq-cons*)

lemma *seq-split*:
  assumes $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$ $rs = rs_1$@rs_2
  obtains t' **where** $\Gamma,\gamma,p\vdash \langle rs_1,s\rangle \Rightarrow t'$ $\Gamma,\gamma,p\vdash \langle rs_2,t'\rangle \Rightarrow t$
  using *assms*
  proof (*induction rs s t arbitrary*: rs_1 rs_2 *thesis rule*: *iptables-bigstep-induct*)
    case *Allow* **thus** *?case* **by** (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros*)
  next
    case *Deny* **thus** *?case* **by** (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros*)
  next
    case *Log* **thus** *?case* **by** (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros*)
  next
    case *Nomatch* **thus** *?case* **by** (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros*)
  next
    case (*Seq rs rsa rsb t t′*)
    hence *rs*: *rsa* @ *rsb* = rs_1 @ rs_2 **by** *simp*
    note *List.append-eq-append-conv-if* [*simp*]
    from *rs* **show** *?case*
      proof (*cases rule*: *list-app-eq-cases*)
        case *longer*
        with *Seq* **have** *t1*: $\Gamma,\gamma,p\vdash \langle$*take* (*length rsa*) rs_1, *Undecided*$\rangle \Rightarrow t$
          by *simp*
        from *Seq longer* **obtain** *t2*
          where *t2a*: $\Gamma,\gamma,p\vdash \langle$*drop* (*length rsa*) rs_1,$t\rangle \Rightarrow$ *t2*
            and *rs2-t2*: $\Gamma,\gamma,p\vdash \langle rs_2,$*t2*$\rangle \Rightarrow t'$
          by *blast*
          with *t1 rs2-t2* **have** $\Gamma,\gamma,p\vdash \langle$*take* (*length rsa*) rs_1 @ *drop* (*length rsa*) rs_1,*Undecided*$\rangle \Rightarrow$ *t2*
          by (*blast intro*: *iptables-bigstep.seq*)
        with *Seq rs2-t2* **show** *?thesis*
          by *simp*

next
case *shorter*
with *rs* **have** *rsa'*: $rsa = rs_1$ @ *take* (*length rsa* − *length* rs_1) rs_2
by (*metis append-eq-conv-conj length-drop*)
from *shorter rs* **have** *rsb'*: rsb = *drop* (*length rsa* − *length* rs_1) rs_2
by (*metis append-eq-conv-conj length-drop*)
from *Seq rsa'* **obtain** *t1*
where *t1a*: Γ,γ,p⊢ ⟨rs_1,*Undecided*⟩ ⇒ *t1*
and *t1b*: Γ,γ,p⊢ ⟨*take* (*length rsa* − *length* rs_1) rs_2,*t1*⟩ ⇒ *t*
by *blast*
from *rsb' Seq.hyps* **have** *t2*: Γ,γ,p⊢ ⟨*drop* (*length rsa* − *length* rs_1) rs_2,*t*⟩ ⇒ *t'*
by *blast*
with *seq' t1b* **have** Γ,γ,p⊢ ⟨rs_2,*t1*⟩ ⇒ *t'*
by *fastforce*
with *Seq t1a* **show** *?thesis*
by *fast*
qed
next
case *Call-return*
hence Γ,γ,p⊢ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* Γ,γ,p⊢ ⟨rs_2, *Undecided*⟩ ⇒ *Undecided*
by (*case-tac* [!] rs_1) (*auto intro*: *iptables-bigstep.skip iptables-bigstep.call-return*)
thus *?case* **by** *fact*
next
case (*Call-result - - - - t*)
show *?case*
proof (*cases* rs_1)
case *Nil*
with *Call-result* **have** Γ,γ,p⊢ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* Γ,γ,p⊢ ⟨rs_2, *Undecided*⟩ ⇒ *t*
by (*auto intro*: *iptables-bigstep.intros*)
thus *?thesis* **by** *fact*
next
case *Cons*
with *Call-result* **have** Γ,γ,p⊢ ⟨rs_1, *Undecided*⟩ ⇒ *t* Γ,γ,p⊢ ⟨rs_2, *t*⟩ ⇒ *t*
by (*auto intro*: *iptables-bigstep.intros*)
thus *?thesis* **by** *fact*
qed
qed (*auto intro*: *iptables-bigstep.intros*)

lemma *seqE*:
assumes Γ,γ,p⊢ ⟨rs_1@rs_2, *s*⟩ ⇒ *t*
obtains *ti* **where** Γ,γ,p⊢ ⟨rs_1,*s*⟩ ⇒ *ti* Γ,γ,p⊢ ⟨rs_2,*ti*⟩ ⇒ *t*
using *assms* **by** (*force elim*: *seq-split*)

lemma *seqE-cons*:
assumes Γ,γ,p⊢ ⟨*r*#*rs*, *s*⟩ ⇒ *t*
obtains *ti* **where** Γ,γ,p⊢ ⟨[*r*],*s*⟩ ⇒ *ti* Γ,γ,p⊢ ⟨*rs*,*ti*⟩ ⇒ *t*

using *assms* **by** (*metis append-Cons append-Nil seqE*)

lemma *nomatch'*:
assumes $\bigwedge r.\ r \in set\ rs \Longrightarrow \neg\ matches\ \gamma\ (get\text{-}match\ r)\ p$
shows $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow s$
proof(*cases s*)
case *Undecided*
have $\forall\, r{\in}set\ rs.\ \neg\ matches\ \gamma\ (get\text{-}match\ r)\ p \Longrightarrow \Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Undecided$
proof(*induction rs*)
case *Nil*
thus *?case* **by** (*fast intro*: *skip*)
next
case (*Cons r rs*)
hence $\Gamma,\gamma,p\vdash \langle [r],\ Undecided\rangle \Rightarrow Undecided$
by (*cases r*) (*auto intro*: *nomatch*)
with *Cons* **show** *?case*
by (*fastforce intro*: *seq-cons*)
qed
with *assms Undecided* **show** *?thesis* **by** *simp*
qed (*blast intro*: *decision*)

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

1. the firewall is in a Decision state

2. the return does not match

In both cases, it is not applied!

lemma *no-free-return*: **assumes** $\Gamma,\gamma,p\vdash \langle [Rule\ m\ Return],\ Undecided\rangle \Rightarrow t$ **and** $matches\ \gamma\ m\ p$ **shows** *False*
proof −
{ fix *a s*
have *no-free-return-hlp*: $\Gamma,\gamma,p\vdash \langle a,s\rangle \Rightarrow t \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow s = Undecided \Longrightarrow a = [Rule\ m\ Return] \Longrightarrow False$
proof (*induction rule*: *iptables-bigstep.induct*)
case (*seq* rs_1)
thus *?case*
by (*cases* rs_1) (*auto dest*: *skipD*)
qed *simp-all*
} with *assms* **show** *?thesis* **by** *blast*
qed

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(*induction arbitrary*: $rs_1\ rs_2\ t'$ *rule*: *iptables-bigstep-induct*)
case *Allow*

thus *?case*
by (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros dest*: *iptables-bigstepD*)
next
case *Deny*
thus *?case*
by (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros dest*: *iptables-bigstepD*)
next
case *Log*
thus *?case*
by (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros dest*: *iptables-bigstepD*)
next
case *Nomatch*
thus *?case*
by (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros dest*: *iptables-bigstepD*)
next
case *Decision*
thus *?case*
by (*cases* rs_1) (*auto intro*: *iptables-bigstep.intros dest*: *iptables-bigstepD*)
next
case(*Seq rs rsa rsb t* t' rs_1 rs_2 t'')
hence *rs*: $rsa @ rsb = rs_1 @ rs_2$ **by** *simp*
note *List.append-eq-append-conv-if* [*simp*]

from *rs* **show** $\Gamma,\gamma,p\vdash \langle rs_2,t''\rangle \Rightarrow t'$
proof(*cases rule*: *list-app-eq-cases*)
case *longer*
have $rs_1 = take\ (length\ rsa)\ rs_1 @ drop\ (length\ rsa)\ rs_1$
by *auto*
with *Seq longer* **show** *?thesis*
by (*metis append-Nil2 skipD seq-split*)
next
case *shorter*
with *Seq*(*7*) *Seq.hyps*(*3*) *Seq.IH*(*1*) *rs* **show** *?thesis*
by (*metis* seq' *append-eq-conv-conj*)
qed
next
case(*Call-return m a chain rsa* m' *rsb*)
have *xx*: $\Gamma,\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow t' \Longrightarrow matches\ \gamma\ m\ p$
$\Longrightarrow$
$\Gamma\ chain = Some\ (rsa @ Rule\ m'\ Return\ \#\ rsb) \Longrightarrow$
$matches\ \gamma\ m'\ p \Longrightarrow$
$\Gamma,\gamma,p\vdash \langle rsa,\ Undecided\rangle \Rightarrow Undecided \Longrightarrow$
$t' = Undecided$
apply(*erule callD*)
apply(*simp-all*)
apply(*erule seqE*)
apply(*erule seqE-cons*)
by (*metis Call-return.IH no-free-return self-append-conv skipD*)

show *?case*
proof (*cases* rs_1)
case (*Cons r rs*)
thus *?thesis*
using *Call-return*
apply(*case-tac* [*Rule m a*] = rs_2)
apply(*simp*)
apply(*simp*)
using *xx* **by** *blast*
next
case *Nil*
moreover hence $t' =$ *Undecided*
by (*metis Call-return.hyps*(*1*) *Call-return.prems*(*2*) *append.simps*(*1*) *decision no-free-return seq state.exhaust*)
moreover have $\bigwedge m.\ \Gamma,\gamma,p\vdash \langle$[*Rule m a*], *Undecided*$\rangle \Rightarrow$ *Undecided*
by (*metis* (*no-types*) *Call-return*(*2*) *Call-return.hyps*(*3*) *Call-return.hyps*(*4*) *Call-return.hyps*(*5*) *call-return nomatch*)
ultimately show *?thesis*
using *Call-return.prems*(*1*) **by** *auto*
qed
next
case(*Call-result m a chain rs t*)
thus *?case*
proof (*cases* rs_1)
case *Cons*
thus *?thesis*
using *Call-result*
apply(*auto simp add*: *iptables-bigstep.skip iptables-bigstep.call-result dest*: *skipD*)
apply(*drule callD*, *simp-all*)
apply *blast*
by (*metis Cons-eq-appendI append-self-conv2 no-free-return seq-split*)
qed (*fastforce intro*: *iptables-bigstep.intros dest*: *skipD*)
qed (*auto dest*: *iptables-bigstepD*)

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 −
{ **fix** *r1 r2 m t*
assume *a1*: $\Gamma,\gamma,p\vdash \langle$*r1* @ *Rule m Return* # *r2*, *Undecided*$\rangle \Rightarrow t$ **and** *a2*: *matches* γ *m p* **and** *a3*: $\Gamma,\gamma,p\vdash \langle$*r1*,*Undecided*$\rangle \Rightarrow$ *Undecided*
have *False*
proof −
from *a1 a3* **have** $\Gamma,\gamma,p\vdash \langle$*Rule m Return* # *r2*, *Undecided*$\rangle \Rightarrow t$
by (*blast intro*: *seq-progress*)
hence $\Gamma,\gamma,p\vdash \langle$[*Rule m Return*] @ *r2*, *Undecided*$\rangle \Rightarrow t$
by *simp*

from *seqE*[*OF this*] **obtain** *ti* **where** $\Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Return}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{ti}$ **by** *blast*
with *no-free-return a2* **show** *False* **by** *fast*
qed
} **note** *no-free-return-seq=this*

from *assms* **show** *?thesis*
proof (*induction arbitrary*: t' *rule*: *iptables-bigstep-induct*)
case *Seq*
thus *?case*
by (*metis seq-progress*)
next
case *Call-result*
thus *?case*
by (*metis no-free-return-seq callD*)
next
case *Call-return*
thus *?case*
by (*metis append-Cons callD no-free-return-seq*)
qed (*auto dest*: *iptables-bigstepD*)
qed

lemma *iptables-bigstep-to-undecided*: $\Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow \mathit{Undecided} \Longrightarrow s = \mathit{Undecided}$
by (*metis decisionD state.exhaust*)

lemma *iptables-bigstep-to-decision*: $\Gamma,\gamma,p\vdash \langle rs,\ \mathit{Decision}\ Y\rangle \Rightarrow \mathit{Decision}\ X \Longrightarrow Y = X$
by (*metis decisionD state.inject*)

lemma *Rule-UndecidedE*:
assumes $\Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ a],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}$
obtains (*nomatch*) $\neg$ *matches* γ *m p*
| (*log*) $a = \mathit{Log} \vee a = \mathit{Empty}$
| (*call*) *c* **where** $a = \mathit{Call}\ c$ *matches* γ *m p*
using *assms*
proof (*induction* [*Rule m a*] *Undecided Undecided rule*: *iptables-bigstep-induct*)
case *Seq*
thus *?case*
by (*metis append-eq-Cons-conv append-is-Nil-conv iptables-bigstep-to-undecided*)
qed *simp-all*

lemma *Rule-DecisionE*:
assumes $\Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ a],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ X$
obtains (*call*) *chain* **where** *matches* γ *m p* $a = \mathit{Call}\ \mathit{chain}$
| (*accept-reject*) *matches* γ *m p* $X = \mathit{FinalAllow} \Longrightarrow a = \mathit{Accept}\ X = \mathit{FinalDeny} \Longrightarrow a = \mathit{Drop} \vee a = \mathit{Reject}$
using *assms*
proof (*induction* [*Rule m a*] *Undecided Decision X rule*: *iptables-bigstep-induct*)
case (*Seq* rs_1)

thus *?case*
by (*cases* rs_1) (*auto dest*: *skipD*)
qed *simp-all*

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 −
from *assms* **obtain** t' **where** t': $\Gamma,\gamma,p\vdash \langle rs_1,\ s\rangle \Rightarrow t'$ $\Gamma,\gamma,p\vdash \langle$[*Rule m Log*] @ $rs_2,\ t'\rangle \Rightarrow t$
by (*blast elim*: *seqE*)
hence $\Gamma,\gamma,p\vdash \langle$*Rule m Log* # $rs_2,\ t'\rangle \Rightarrow t$
by *simp*
then obtain t'' **where** $\Gamma,\gamma,p\vdash \langle$[*Rule m Log*], $t'\rangle \Rightarrow t''$ $\Gamma,\gamma,p\vdash \langle rs_2,\ t''\rangle \Rightarrow t$
by (*blast elim*: *seqE-cons*)
with t' **show** *?thesis*
by (*metis state.exhaust iptables-bigstep-deterministic decision log nomatch seq*)
qed
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 −
from *assms* **obtain** t' **where** t': $\Gamma,\gamma,p\vdash \langle rs_1,\ s\rangle \Rightarrow t'$ $\Gamma,\gamma,p\vdash \langle$[*Rule m Empty*] @ $rs_2,\ t'\rangle \Rightarrow t$
by (*blast elim*: *seqE*)
hence $\Gamma,\gamma,p\vdash \langle$*Rule m Empty* # $rs_2,\ t'\rangle \Rightarrow t$
by *simp*
then obtain t'' **where** $\Gamma,\gamma,p\vdash \langle$[*Rule m Empty*], $t'\rangle \Rightarrow t''$ $\Gamma,\gamma,p\vdash \langle rs_2,\ t''\rangle \Rightarrow t$
by (*blast elim*: *seqE-cons*)
with t' **show** *?thesis*
by (*metis state.exhaust iptables-bigstep-deterministic decision empty nomatch seq*)
qed

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

locale *iptables-bigstep-fixedbackground* =
fixes Γ::*'a ruleset*
and γ::(*'a*, *'p*) *matcher*
begin

inductive *iptables-bigstep'* :: *'p* ⇒ *'a rule list* ⇒ *state* ⇒ *state* ⇒ *bool*
(-⊢' ⟨-, -⟩ ⇒ - [*60*,*20*,*98*,*98*] *89*)
for *p* **where**
skip: $p\vdash' \langle[],\ t\rangle \Rightarrow t$ |
accept: *matches* γ *m p* $\Longrightarrow$ $p\vdash' \langle$[*Rule m Accept*], *Undecided*⟩ ⇒ *Decision Fi-*

nalAllow |
drop: *matches* γ *m p* $\Longrightarrow$ *p*$\vdash'$ $\langle$[*Rule m Drop*], *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalDeny* |
reject: *matches* γ *m p* $\Longrightarrow$ *p*$\vdash'$ $\langle$[*Rule m Reject*], *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalDeny* |
log: *matches* γ *m p* $\Longrightarrow$ *p*$\vdash'$ $\langle$[*Rule m Log*], *Undecided*$\rangle$ $\Rightarrow$ *Undecided* |
empty: *matches* γ *m p* $\Longrightarrow$ *p*$\vdash'$ $\langle$[*Rule m Empty*], *Undecided*$\rangle$ $\Rightarrow$ *Undecided* |
nomatch: $\neg$ *matches* γ *m p* $\Longrightarrow$ *p*$\vdash'$ $\langle$[*Rule m a*], *Undecided*$\rangle$ $\Rightarrow$ *Undecided* |
decision: *p*$\vdash'$ $\langle$*rs*, *Decision X*$\rangle$ $\Rightarrow$ *Decision X* |
seq: $[\![$*p*$\vdash'$ $\langle rs_1$, *Undecided*$\rangle$ $\Rightarrow$ *t*; *p*$\vdash'$ $\langle rs_2$, *t*$\rangle$ $\Rightarrow$ *t'*$]\!]$ $\Longrightarrow$ *p*$\vdash'$ $\langle rs_1@rs_2$, *Undecided*$\rangle$ $\Rightarrow$ *t'* |
call-return: $[\![$ *matches* γ *m p*; Γ *chain* = *Some* (rs_1@[*Rule m' Return*]@rs_2);
matches γ *m' p*; *p*$\vdash'$ $\langle rs_1$, *Undecided*$\rangle$ $\Rightarrow$ *Undecided* $]\!]$ $\Longrightarrow$
p$\vdash'$ $\langle$[*Rule m* (*Call chain*)], *Undecided*$\rangle$ $\Rightarrow$ *Undecided* |
call-result: $[\![$ *matches* γ *m p*; *p*$\vdash'$ $\langle$*the* (Γ *chain*), *Undecided*$\rangle$ $\Rightarrow$ *t* $]\!]$ $\Longrightarrow$
p$\vdash'$ $\langle$[*Rule m* (*Call chain*)], *Undecided*$\rangle$ $\Rightarrow$ *t*

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

lemma *wf*-Γ-*append*: *wf*-Γ (*rs1@rs2*) $\longleftrightarrow$ *wf*-Γ *rs1* $\wedge$ *wf*-Γ *rs2*
by(*simp add*: *wf*-Γ-*def*, *blast*)
lemma *wf*-Γ-*tail*: *wf*-Γ (*r* # *rs*) $\Longrightarrow$ *wf*-Γ *rs* **by**(*simp add*: *wf*-Γ-*def*)
lemma *wf*-Γ-*Call*: *wf*-Γ [*Rule m* (*Call chain*)] $\Longrightarrow$ *wf*-Γ (*the* (Γ *chain*)) $\wedge$ ($\exists$ *rs*. Γ *chain* = *Some rs*)
apply(*simp add*: *wf*-Γ-*def*)
by (*metis option.collapse ranI*)

lemma *wf*-Γ *rs* $\Longrightarrow$ *p*$\vdash'$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow$ *t* $\longleftrightarrow$ Γ,γ,*p*$\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow$ *t*
apply(*rule iffI*)
apply(*rotate-tac 1*)
apply(*induction rs s t rule*: *iptables-bigstep'.induct*)
apply(*auto intro*: *iptables-bigstep.intros simp*: *wf*-Γ-*append dest*!: *wf*-Γ-*Call*)[*11*]
apply(*rotate-tac 1*)
apply(*induction rs s t rule*: *iptables-bigstep.induct*)
apply(*auto intro*: *iptables-bigstep'.intros simp*: *wf*-Γ-*append dest*!: *wf*-Γ-*Call*)[*11*]
done

end

end
theory *Matching*
imports *Semantics*
begin

2.1 Boolean Matcher Algebra

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

lemma *matches-rule-iptables-bigstep*:
assumes *matches* γ *m p* $\longleftrightarrow$ *matches* γ *m' p*
shows $\Gamma,\gamma,p\vdash \langle[$*Rule m a*$], s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle[$*Rule m' a*$], s\rangle \Rightarrow t$ (**is** *?l* $\longleftrightarrow$*?r*)
proof –
{
fix *m m'*
assume $\Gamma,\gamma,p\vdash \langle[$*Rule m a*$], s\rangle \Rightarrow t$ *matches* γ *m p* $\longleftrightarrow$ *matches* γ *m' p*
hence $\Gamma,\gamma,p\vdash \langle[$*Rule m' a*$], s\rangle \Rightarrow t$
by (*induction* [*Rule m a*] *s t rule*: *iptables-bigstep-induct*)
(*auto intro*: *iptables-bigstep.intros simp*: *Cons-eq-append-conv dest*: *skipD*)
}
with *assms* **show** *?thesis* **by** *blast*
qed

lemma *matches-rule-and-simp-help*:
assumes *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash \langle[$*Rule* (*MatchAnd m m'*) *a'*$]$, *Undecided*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle[$*Rule m' a'*$]$, *Undecided*$\rangle \Rightarrow t$ (**is** *?l* $\longleftrightarrow$*?r*)
proof
assume *?l* **thus** *?r*
by (*induction* [*Rule* (*MatchAnd m m'*) *a'*] *Undecided t rule*: *iptables-bigstep-induct*)
(*auto intro*: *iptables-bigstep.intros simp*: *assms Cons-eq-append-conv dest*: *skipD*)
next
assume *?r* **thus** *?l*
by (*induction* [*Rule m' a'*] *Undecided t rule*: *iptables-bigstep-induct*)
(*auto intro*: *iptables-bigstep.intros simp*: *assms Cons-eq-append-conv dest*: *skipD*)
qed

lemma *matches-MatchNot-simp*:
assumes *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash \langle[$*Rule* (*MatchNot m*) *a*$]$, *Undecided*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle[]$, *Undecided*$\rangle \Rightarrow t$ (**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *?l* **thus** *?r*
by (*induction* [*Rule* (*MatchNot m*) *a*] *Undecided t rule*: *iptables-bigstep-induct*)
(*auto intro*: *iptables-bigstep.intros simp*: *assms Cons-eq-append-conv dest*: *skipD*)
next
assume *?r*
hence $t =$ *Undecided*
by (*metis skipD*)
with *assms* **show** *?l*
by (*fastforce intro*: *nomatch*)
qed

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
  assume *?l* **thus** *?r*
   by (*induction* [*Rule* (*MatchAnd* (*MatchNot m*) *m'*) *a*] *Undecided t rule*: *iptables-bigstep-induct*)
      (*auto intro*: *iptables-bigstep.intros simp add*: *assms Cons-eq-append-conv dest*:
skipD)
next
  assume *?r*
  hence t = *Undecided*
    by (*metis skipD*)
  with *assms* **show** *?l*
    by (*fastforce intro*: *nomatch*)
qed

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 (*cases s*)
  case *Undecided*
  with *assms* **show** *?thesis*
    by (*simp add*: *matches-rule-and-simp-help*)
next
  case *Decision*
  thus *?thesis* **by** (*metis decision decisionD*)
qed

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 –
  { fix *m1 m2*
   have $\Gamma,\gamma,p\vdash \langle[$*Rule* (*MatchAnd m1 m2*) *a*$]$, $s\rangle \Rightarrow t \Longrightarrow \Gamma,\gamma,p\vdash \langle[$*Rule* (*MatchAnd*
m2 m1) *a*$]$, $s\rangle \Rightarrow t$
     proof (*induction* [*Rule* (*MatchAnd m1 m2*) *a*] *s t rule*: *iptables-bigstep-induct*)
        case *Seq* **thus** *?case*
          by (*metis Nil-is-append-conv append-Nil butlast-append butlast-snoc seq*)
      qed (*auto intro*: *iptables-bigstep.intros*)
  }
  thus *?thesis* **by** *blast*
qed

definition *add-match* :: *'a match-expr* $\Rightarrow$ *'a rule list* $\Rightarrow$ *'a rule list* **where**
  add-match m rs = *map* ($\lambda r.$ *case r 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*
  unfolding *add-match-def*
  by (*fact map-append*)

lemma *add-match-split-fst*: *add-match m* (*Rule m' a'* # *rs*) = *Rule* (*MatchAnd m m'*) *a'* # *add-match m rs*
  unfolding *add-match-def*
  by *simp*

lemma *matches-add-match-simp*:
  assumes *m*: *matches* γ *m p*
  shows $\Gamma,\gamma,p\vdash$ ⟨*add-match m rs*, *s*⟩ ⇒ *t* ⟷ $\Gamma,\gamma,p\vdash$ ⟨*rs*, *s*⟩ ⇒ *t* (**is** *?l* ⟷ *?r*)
  proof
    assume *?l* **with** *m* **show** *?r*
      proof (*induction rs*)
        case *Nil*
        thus *?case*
          unfolding *add-match-def* **by** *simp*
      next
        case (*Cons r rs*)
        thus *?case*
          apply(*cases r*)
          apply(*simp only*: *add-match-split-fst*)
          apply(*erule seqE-cons*)
          apply(*simp only*: *matches-rule-and-simp*)
          apply(*metis decision state.exhaust iptables-bigstep-deterministic seq-cons*)
          done
      qed
  next
    assume *?r* **with** *m* **show** *?l*
      proof (*induction rs*)
        case *Nil*
        thus *?case*
          unfolding *add-match-def* **by** *simp*
      next
        case (*Cons r rs*)
        thus *?case*
          apply(*cases r*)
          apply(*simp only*: *add-match-split-fst*)
          apply(*erule seqE-cons*)
          apply(*subst*(*asm*) *matches-rule-and-simp*[*symmetric*])
          apply(*simp*)
          apply(*metis decision state.exhaust iptables-bigstep-deterministic seq-cons*)
          done
      qed

qed

lemma *matches-add-match-MatchNot-simp*:
assumes *m*: *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash$ ⟨*add-match* (*MatchNot m*) *rs*, *s*⟩ $\Rightarrow$ *t* $\longleftrightarrow$ $\Gamma,\gamma,p\vdash$ ⟨[], *s*⟩ $\Rightarrow$ *t* (**is** *?l s* $\longleftrightarrow$ *?r s*)
proof (*cases s*)
case *Undecided*
have *?l Undecided* $\longleftrightarrow$ *?r Undecided*
proof
assume *?l Undecided* **with** *m* **show** *?r Undecided*
proof (*induction rs*)
case *Nil*
thus *?case*
unfolding *add-match-def* **by** *simp*
next
case (*Cons r rs*)
thus *?case*
by (*cases r*) (*metis matches-MatchNotAnd-simp skipD seqE-cons add-match-split-fst*)
qed
next
assume *?r Undecided* **with** *m* **show** *?l Undecided*
proof (*induction rs*)
case *Nil*
thus *?case*
unfolding *add-match-def* **by** *simp*
next
case (*Cons r rs*)
thus *?case*
by (*cases r*) (*metis matches-MatchNotAnd-simp skipD seq'-cons add-match-split-fst*)
qed
qed
with *Undecided* **show** *?thesis* **by** *fast*
next
case (*Decision d*)
thus *?thesis*
by(*metis decision decisionD*)
qed

lemma *not-matches-add-match-simp*:
assumes $\neg$ *matches* γ *m p*
shows $\Gamma,\gamma,p\vdash$ ⟨*add-match m rs*, *Undecided*⟩ $\Rightarrow$ *t* $\longleftrightarrow$ $\Gamma,\gamma,p\vdash$ ⟨[], *Undecided*⟩ $\Rightarrow$ *t*
proof(*induction rs*)
case *Nil*
thus *?case*
unfolding *add-match-def* **by** *simp*

next
case (*Cons r rs*)
thus *?case*
by (*cases r*) (*metis assms add-match-split-fst matches.simps*(*1*) *nomatch seq'-cons nomatchD seqE-cons*)
qed

lemma *iptables-bigstep-add-match-notnot-simp*:
$\Gamma,\gamma,p\vdash$ ⟨*add-match* (*MatchNot* (*MatchNot m*)) *rs*, *s*⟩ ⇒ *t* ⟷ $\Gamma,\gamma,p\vdash$ ⟨*add-match m rs*, *s*⟩ ⇒ *t*
proof(*induction rs*)
case *Nil*
thus *?case*
unfolding *add-match-def* **by** *simp*
next
case (*Cons r rs*)
thus *?case*
by (*cases r*)
(*metis decision decisionD state.exhaust matches.simps*(*2*) *matches-add-match-simp not-matches-add-match-simp*)
qed

lemma *not-matches-add-matchNot-simp*:
¬ *matches γ m p* ⟹ $\Gamma,\gamma,p\vdash$ ⟨*add-match* (*MatchNot m*) *rs*, *s*⟩ ⇒ *t* ⟷ $\Gamma,\gamma,p\vdash$ ⟨*rs*, *s*⟩ ⇒ *t*
by (*simp add*: *matches-add-match-simp*)

lemma *iptables-bigstep-add-match-and*:
$\Gamma,\gamma,p\vdash$ ⟨*add-match m1* (*add-match m2 rs*), *s*⟩ ⇒ *t* ⟷ $\Gamma,\gamma,p\vdash$ ⟨*add-match* (*MatchAnd m1 m2*) *rs*, *s*⟩ ⇒ *t*
proof(*induction rs arbitrary*: *s t*)
case *Nil*
thus *?case*
unfolding *add-match-def* **by** *simp*
next
case(*Cons r rs*)
show *?case*
proof (*cases r*, *simp only*: *add-match-split-fst*)
fix *m a*
show $\Gamma,\gamma,p\vdash$ ⟨*Rule* (*MatchAnd m1* (*MatchAnd m2 m*)) *a* # *add-match m1* (*add-match m2 rs*), *s*⟩ ⇒ *t* ⟷ $\Gamma,\gamma,p\vdash$ ⟨*Rule* (*MatchAnd* (*MatchAnd m1 m2*) *m*) *a* # *add-match* (*MatchAnd m1 m2*) *rs*, *s*⟩ ⇒ *t* (**is** *?l* ⟷ *?r*)
proof
assume *?l* **with** *Cons.IH* **show** *?r*
apply −
apply(*erule seqE-cons*)
apply(*case-tac s*)
apply(*case-tac ti*)
apply (*metis matches.simps*(*1*) *matches-rule-and-simp matches-rule-and-simp-help*

nomatch seq′-cons)
apply (*metis add-match-split-fst matches.simps*(*1*) *matches-add-match-simp not-matches-add-match-simp seq-cons*)
apply (*metis decision decisionD*)
done
next
assume *?r* **with** *Cons.IH* **show** *?l*
apply −
apply(*erule seqE-cons*)
apply(*case-tac s*)
apply(*case-tac ti*)
apply (*metis matches.simps*(*1*) *matches-rule-and-simp matches-rule-and-simp-help nomatch seq′-cons*)
apply (*metis add-match-split-fst matches.simps*(*1*) *matches-add-match-simp not-matches-add-match-simp seq-cons*)
apply (*metis decision decisionD*)
done
qed
qed
qed

end
theory *Call-Return-Unfolding*
imports *Matching*
begin

3 *Call Return* Unfolding

Remove *Return*s

fun *process-ret* :: *′a rule list* ⇒ *′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 *Call*s

fun *process-call* :: *′a ruleset* ⇒ *′a rule list* ⇒ *′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 = *Return* ⟹ *process-ret* (*Rule m a* # *rs*) = *add-match* (*MatchNot m*) (*process-ret rs*)
by *auto*

lemma *process-ret-split-fst-NeqReturn*:
a ≠ *Return* ⟹ *process-ret*((*Rule m a*) # *rs*) = (*Rule m a*) # (*process-ret rs*)

by (*cases a*) *auto*

lemma *add-match-simp*: *add-match m* = *map* (λ*r*. *Rule* (*MatchAnd m* (*get-match r*)) (*get-action r*))
by (*auto simp*: *add-match-def cong*: *map-cong split*: *rule.split*)

definition *add-missing-ret-unfoldings* :: *'a rule list* ⇒ *'a rule list* ⇒ *'a rule list*
where
add-missing-ret-unfoldings rs1 rs2 ≡
foldr (λ*rf acc*. *add-match* (*MatchNot* (*get-match rf*)) ∘ *acc*) [*r*←*rs1*. *get-action r* = *Return*] *id rs2*

fun *MatchAnd-foldr* :: *'a match-expr list* ⇒ *'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* ⇒ (*'a rule list* ⇒ *'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*:
Γ,γ,*p*⊢ ⟨*add-match m* (*add-match-MatchAnd-foldr ms rs2*), *s*⟩ ⇒ *t* = Γ,γ,*p*⊢ ⟨*add-match* (*MatchAnd-foldr* (*m*#*ms*)) *rs2*, *s*⟩ ⇒ *t*
proof (*induction ms*)
case *Nil*
show *?case* **by** (*simp add*: *add-match-def*)
next
case *Cons*
thus *?case* **by** (*simp add*: *iptables-bigstep-add-match-and*)
qed

lemma *add-match-MatchAnd-foldr-empty-rs2*: *add-match-MatchAnd-foldr ms* [] = []
by (*induction ms*) (*simp-all add*: *add-match-def*)

lemma *add-missing-ret-unfoldings-alt*: Γ,γ,*p*⊢ ⟨*add-missing-ret-unfoldings rs1 rs2*, *s*⟩ ⇒ *t* ⟷
Γ,γ,*p*⊢ ⟨(*add-match-MatchAnd-foldr* (*map* (λ*r*. *MatchNot* (*get-match r*)) [*r*←*rs1*. *get-action r* = *Return*])) *rs2*, *s* ⟩ ⇒ *t*
proof(*induction rs1*)
case *Nil*
thus *?case*
unfolding *add-missing-ret-unfoldings-def* **by** *simp*
next
case (*Cons r rs*)
from *Cons* **obtain** *m a* **where** *r* = *Rule m a* **by**(*cases r*) (*simp*)
with *Cons* **show** *?case*

unfolding *add-missing-ret-unfoldings-def*
apply(*cases matches* γ *m p*)
apply (*simp-all add*: *matches-add-match-simp matches-add-match-MatchNot-simp add-match-add-match-MatchAnd-foldr*[*symmetric*])
done
qed

lemma *add-match-add-missing-ret-unfoldings-rot*:
$\Gamma,\gamma,p\vdash \langle$*add-match m* (*add-missing-ret-unfoldings rs1 rs2*), $s\rangle \Rightarrow t$ =
$\Gamma,\gamma,p\vdash \langle$*add-missing-ret-unfoldings* (*Rule* (*MatchNot m*) *Return#rs1*) *rs2*, $s\rangle \Rightarrow t$
by(*simp add*: *add-missing-ret-unfoldings-def iptables-bigstep-add-match-notnot-simp*)

3.1 Completeness

lemma *process-ret-split-obvious*: *process-ret* (rs_1 @ rs_2) =
(*process-ret* rs_1) @ (*add-missing-ret-unfoldings* rs_1 (*process-ret* rs_2))
unfolding *add-missing-ret-unfoldings-def*
proof (*induction* rs_1 *arbitrary*: rs_2)
case (*Cons r rs*)
from *Cons* **obtain** *m a* **where** *r* = *Rule m a* **by** (*cases r*) *simp*
with *Cons.IH* **show** *?case*
apply(*cases a*)
apply(*simp-all add*: *add-match-split*)
done
qed *simp*

lemma *add-match-distrib*:
$\Gamma,\gamma,p\vdash \langle$*add-match m1* (*add-match m2 rs*), $s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle$*add-match m2* (*add-match m1 rs*), $s\rangle \Rightarrow t$
proof –
{
fix *m1 m2*
have $\Gamma,\gamma,p\vdash \langle$*add-match m1* (*add-match m2 rs*), $s\rangle \Rightarrow t \Longrightarrow \Gamma,\gamma,p\vdash \langle$*add-match m2* (*add-match m1 rs*), $s\rangle \Rightarrow t$
proof (*induction rs arbitrary*: *s*)
case *Nil* **thus** *?case* **by** (*simp add*: *add-match-def*)
next
case (*Cons r rs*)
from *Cons* **obtain** *m a* **where** *r*: *r* = *Rule m a* **by** (*cases r*) *simp*
with *Cons.prems* **obtain** *ti* **where** *1*: $\Gamma,\gamma,p\vdash \langle$[*Rule* (*MatchAnd m1* (*MatchAnd m2 m*)) *a*], $s\rangle \Rightarrow ti$ **and** *2*: $\Gamma,\gamma,p\vdash \langle$*add-match m1* (*add-match m2 rs*), $ti\rangle \Rightarrow t$
apply(*simp add*: *add-match-split-fst*)
apply(*erule seqE-cons*)
by *simp*
from *1 r* **have** *base*: $\Gamma,\gamma,p\vdash \langle$[*Rule* (*MatchAnd m2* (*MatchAnd m1 m*)) *a*], $s\rangle \Rightarrow ti$
by (*metis matches.simps*(*1*) *matches-rule-iptables-bigstep*)

from *2 Cons.IH* **have** *IH*: $\Gamma,\gamma,p\vdash$ $\langle$*add-match m2* (*add-match m1 rs*), *ti*$\rangle$ $\Rightarrow$ *t* **by** *simp*
from *base IH seq'-cons* **have** $\Gamma,\gamma,p\vdash$ $\langle$*Rule* (*MatchAnd m2* (*MatchAnd m1 m*)) *a* # *add-match m2* (*add-match m1 rs*), *s*$\rangle$ $\Rightarrow$ *t* **by** *fast*
thus *?case* **using** *r* **by**(*simp add*: *add-match-split-fst*[*symmetric*])
qed
}
thus *?thesis* **by** *blast*
qed

lemma *add-missing-ret-unfoldings-emptyrs2*: *add-missing-ret-unfoldings rs1* [] = []
unfolding *add-missing-ret-unfoldings-def*
by (*induction rs1*) (*simp-all add*: *add-match-def*)

lemma *process-call-split*: *process-call* Γ (*rs1* @ *rs2*) = *process-call* Γ *rs1* @ *process-call* Γ *rs2*
proof (*induction rs1*)
case (*Cons r rs1*)
thus *?case*
apply(*cases r*, *rename-tac m a*)
apply(*case-tac a*)
apply(*simp-all*)
done
qed *simp*

lemma *add-match-split-fst'*: *add-match m* (*a* # *rs*) = *add-match m* [*a*] @ *add-match m rs*
by (*simp add*: *add-match-split*[*symmetric*])

lemma *process-call-split-fst*: *process-call* Γ (*a* # *rs*) = *process-call* Γ [*a*] @ *process-call* Γ *rs*
by (*simp add*: *process-call-split*[*symmetric*])

lemma *iptables-bigstep-process-ret-undecided*: $\Gamma,\gamma,p\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *t* $\Longrightarrow$ $\Gamma,\gamma,p\vdash$ $\langle$*process-ret rs*, *Undecided*$\rangle$ $\Rightarrow$ *t*
proof (*induction rs*)
case (*Cons r rs*)
show *?case*
proof (*cases r*)
case (*Rule m' a'*)
show $\Gamma,\gamma,p\vdash$ $\langle$*process-ret* (*r* # *rs*), *Undecided*$\rangle$ $\Rightarrow$ *t*
proof (*cases a'*)
case *Accept*
with *Cons Rule* **show** *?thesis*
by *simp* (*metis acceptD decision decisionD nomatchD seqE-cons seq-cons*)
next
case *Drop*

with *Cons Rule* **show** *?thesis*
by *simp* (*metis decision decisionD dropD nomatchD seqE-cons seq-cons*)
next
case *Log*
with *Cons Rule* **show** *?thesis*
by *simp* (*metis logD nomatchD seqE-cons seq-cons*)
next
case *Reject*
with *Cons Rule* **show** *?thesis*
by *simp* (*metis decision decisionD nomatchD rejectD seqE-cons seq-cons*)
next
case (*Call chain*)
from *Cons.prems* **obtain** *ti* **where** *1*:$\Gamma,\gamma,p\vdash \langle [r],\ Undecided\rangle \Rightarrow ti$ **and** *2*: $\Gamma,\gamma,p\vdash \langle rs,\ ti\rangle \Rightarrow t$ **using** *seqE-cons* **by** *metis*
thus *?thesis*
proof(*cases ti*)
case *Undecided*
with *Cons.IH 2* **have** *IH*: $\Gamma,\gamma,p\vdash \langle process\text{-}ret\ rs,\ Undecided\rangle \Rightarrow t$ **by** *simp*
from *Undecided 1 Call Rule* **have** $\Gamma,\gamma,p\vdash \langle [Rule\ m'\ (Call\ chain)],\ Undecided\rangle \Rightarrow Undecided$ **by** *simp*
with *IH* **have** $\Gamma,\gamma,p\vdash \langle Rule\ m'\ (Call\ chain)\ \#\ process\text{-}ret\ rs,\ Undecided\rangle \Rightarrow t$ **using** *seq′-cons* **by** *fast*
thus *?thesis* **using** *Rule Call* **by** *force*
next
case (*Decision X*)
with *1 Rule Call* **have** $\Gamma,\gamma,p\vdash \langle [Rule\ m'\ (Call\ chain)],\ Undecided\rangle \Rightarrow Decision\ X$ **by** *simp*
moreover from *2 Decision* **have** $t = Decision\ X$ **using** *decisionD* **by** *fast*
moreover from *decision* **have** $\Gamma,\gamma,p\vdash \langle process\text{-}ret\ rs,\ Decision\ X\rangle \Rightarrow Decision\ X$ **by** *fast*
ultimately show *?thesis* **using** *seq-cons* **by** (*metis Call Rule process-ret.simps*(*7*))
qed
next
case *Return*
with *Cons Rule* **show** *?thesis*
by *simp* (*metis matches.simps*(*2*) *matches-add-match-simp no-free-return nomatchD seqE-cons*)
next
case *Empty*
show *?thesis*
apply (*insert Empty Cons Rule*)
apply(*erule seqE-cons*)
apply (*rename-tac ti*)
apply(*case-tac ti*)
apply (*metis process-ret.simps*(*8*) *seq′-cons*)
apply (*metis Rule-DecisionE emptyD state.distinct*(*1*))

done
next
case *Unknown*
show *?thesis*
apply (*insert Unknown Cons Rule*)
apply(*erule seqE-cons*)
apply(*case-tac ti*)
apply (*metis process-ret.simps*(*9*) *seq′-cons*)
apply (*metis decision iptables-bigstep-deterministic process-ret.simps*(*9*) *seq-cons*)
done
qed
qed
qed *simp*

lemma *add-match-rot-add-missing-ret-unfoldings*:
$\Gamma,\gamma,p\vdash$ ⟨*add-match m* (*add-missing-ret-unfoldings rs1 rs2*), *Undecided*⟩ ⇒ *Undecided* =
$\Gamma,\gamma,p\vdash$ ⟨*add-missing-ret-unfoldings rs1* (*add-match m rs2*), *Undecided*⟩ ⇒ *Undecided*
apply(*simp add*: *add-missing-ret-unfoldings-alt add-match-add-missing-ret-unfoldings-rot add-match-add-match-MatchAnd-foldr*[*symmetric*] *iptables-bigstep-add-match-notnot-simp*)
apply(*cases map* (λr. *MatchNot* (*get-match r*)) [*r*←*rs1* . (*get-action r*) = *Return*])
apply(*simp-all add*: *add-match-distrib*)
done

Completeness

theorem *unfolding-complete*: $\Gamma,\gamma,p\vdash$ ⟨*rs*,*s*⟩ ⇒ *t* $\Longrightarrow$ $\Gamma,\gamma,p\vdash$ ⟨*process-call* Γ *rs*,*s*⟩ ⇒ *t*
proof (*induction rule*: *iptables-bigstep-induct*)
case (*Nomatch m a*)
thus *?case*
by (*cases a*) (*auto intro*: *iptables-bigstep.intros simp add*: *not-matches-add-match-simp skip*)
next
case *Seq*
thus *?case*
by(*simp add*: *process-call-split seq′*)
next
case (*Call-return m a chain* rs_1 m' rs_2)
hence $\Gamma,\gamma,p\vdash$ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided*
by *simp*
hence $\Gamma,\gamma,p\vdash$ ⟨*process-ret* rs_1, *Undecided*⟩ ⇒ *Undecided*
by (*rule iptables-bigstep-process-ret-undecided*)
with *Call-return* **have** $\Gamma,\gamma,p\vdash$ ⟨*process-ret* rs_1 @ *add-missing-ret-unfoldings* rs_1 (*add-match* (*MatchNot* m') (*process-ret* rs_2)), *Undecided*⟩ ⇒ *Undecided*
by (*metis matches-add-match-MatchNot-simp skip add-match-rot-add-missing-ret-unfoldings seq′*)
with *Call-return* **show** *?case*

by (*simp add*: *matches-add-match-simp process-ret-split-obvious*)
next
case *Call-result*
thus *?case*
by (*simp add*: *matches-add-match-simp iptables-bigstep-process-ret-undecided*)
qed (*auto intro*: *iptables-bigstep.intros*)

lemma *process-ret-cases*:
process-ret rs = *rs* $\vee$ ($\exists$ rs_1 rs_2 *m*. *rs* = rs_1@[*Rule m Return*]@rs_2 $\wedge$ (*process-ret rs*) = rs_1@(*process-ret* ([*Rule m Return*]@rs_2)))
proof (*induction rs*)
case (*Cons r rs*)
thus *?case*
apply(*cases r*, *rename-tac m′ a′*)
apply(*case-tac a′*)
apply(*simp-all*)
apply(*erule disjE*,*simp*,*rule disjI2*,*elim exE*,*simp add*: *process-ret-split-obvious*, *metis append-Cons process-ret-split-obvious process-ret.simps*(*2*))+
apply(*rule disjI2*)
apply(*rule-tac x*=[] **in** *exI*)
apply(*rule-tac x*=*rs* **in** *exI*)
apply(*rule-tac x*=*m′* **in** *exI*)
apply(*simp*)
apply(*erule disjE*,*simp*,*rule disjI2*,*elim exE*,*simp add*: *process-ret-split-obvious*, *metis append-Cons process-ret-split-obvious process-ret.simps*(*2*))+
done
qed *simp*

lemma *process-ret-splitcases*:
obtains (*id*) *process-ret rs* = *rs*
| (*split*) rs_1 rs_2 *m* **where** *rs* = rs_1@[*Rule m Return*]@rs_2 **and** *process-ret rs* = rs_1@(*process-ret* ([*Rule m Return*]@rs_2))
by (*metis process-ret-cases*)

lemma *iptables-bigstep-process-ret-cases3*:
assumes Γ,γ,$p\vdash$ $\langle$*process-ret rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
obtains (*noreturn*) Γ,γ,$p\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
| (*return*) rs_1 rs_2 *m* **where** *rs* = rs_1@[*Rule m Return*]@rs_2 Γ,γ,$p\vdash$ $\langle rs_1$, *Undecided*$\rangle$ $\Rightarrow$ *Undecided matches* γ *m p*
proof −
have Γ,γ,$p\vdash$ $\langle$*process-ret rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided* $\Longrightarrow$
(Γ,γ,$p\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*) $\vee$
($\exists$ rs_1 rs_2 *m*. *rs* = rs_1@[*Rule m Return*]@rs_2 $\wedge$ Γ,γ,$p\vdash$ $\langle rs_1$, *Undecided*$\rangle$ $\Rightarrow$ *Undecided* $\wedge$ *matches* γ *m p*)
proof (*induction rs*)
case *Nil* **thus** *?case* **by** *simp*

next
case (*Cons r rs*)
from *Cons* **obtain** *m a* **where** *r*: $r = Rule\ m\ a$ **by** (*cases r*) *simp*
from *r Cons* **show** *?case*
proof(*cases* $a \neq Return$)
case *True*
with *r Cons.prems* **have** *prems-r*: $\Gamma,\gamma,p\vdash \langle[Rule\ m\ a],\ Undecided\rangle \Rightarrow Undecided$ **and** *prems-rs*: $\Gamma,\gamma,p\vdash \langle process\text{-}ret\ rs,\ Undecided\rangle \Rightarrow Undecided$
apply(*simp-all add*: *process-ret-split-fst-NeqReturn*)
apply(*erule seqE-cons*, *frule iptables-bigstep-to-undecided*, *simp*)+
done
from *prems-rs Cons.IH* **have** $\Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Undecided \vee (\exists rs_1\ rs_2\ m.\ rs = rs_1\ @\ [Rule\ m\ Return]\ @\ rs_2 \wedge \Gamma,\gamma,p\vdash \langle rs_1,\ Undecided\rangle \Rightarrow Undecided \wedge matches\ \gamma\ m\ p)$ **by** *simp*
thus $\Gamma,\gamma,p\vdash \langle r\ \#\ rs,\ Undecided\rangle \Rightarrow Undecided \vee (\exists rs_1\ rs_2\ m.\ r\ \#\ rs = rs_1\ @\ [Rule\ m\ Return]\ @\ rs_2 \wedge \Gamma,\gamma,p\vdash \langle rs_1,\ Undecided\rangle \Rightarrow Undecided \wedge matches\ \gamma\ m\ p)$ (**is** *?goal*)
proof(*elim disjE*)
assume $\Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Undecided$
hence $\Gamma,\gamma,p\vdash \langle r\ \#\ rs,\ Undecided\rangle \Rightarrow Undecided$ **using** *prems-r* **by** (*metis r seq′-cons*)
thus *?goal* **by** *simp*
next
assume $(\exists rs_1\ rs_2\ m.\ rs = rs_1\ @\ [Rule\ m\ Return]\ @\ rs_2 \wedge \Gamma,\gamma,p\vdash \langle rs_1,\ Undecided\rangle \Rightarrow Undecided \wedge matches\ \gamma\ m\ p)$
from *this* **obtain** $rs_1\ rs_2\ m'$ **where** $rs = rs_1\ @\ [Rule\ m'\ Return]\ @\ rs_2$ **and** $\Gamma,\gamma,p\vdash \langle rs_1,\ Undecided\rangle \Rightarrow Undecided$ **and** $matches\ \gamma\ m'\ p$ **by** *blast*
hence $\exists rs_1\ rs_2\ m.\ r\ \#\ rs = rs_1\ @\ [Rule\ m\ Return]\ @\ rs_2 \wedge \Gamma,\gamma,p\vdash \langle rs_1,\ Undecided\rangle \Rightarrow Undecided \wedge matches\ \gamma\ m\ p$
apply(*rule-tac* $x{=}Rule\ m\ a\ \#\ rs_1$ **in** *exI*)
apply(*rule-tac* $x{=}rs_2$ **in** *exI*)
apply(*rule-tac* $x{=}m'$ **in** *exI*)
apply(*simp add*: *r*)
using *prems-r seq′-cons* **by** *fast*
thus *?goal* **by** *simp*
qed
next
case *False*
hence $a = Return$ **by** *simp*
with *Cons.prems r* **have** *prems*: $\Gamma,\gamma,p\vdash \langle add\text{-}match\ (MatchNot\ m)\ (process\text{-}ret\ rs),\ Undecided\rangle \Rightarrow Undecided$ **by** *simp*
show $\Gamma,\gamma,p\vdash \langle r\ \#\ rs,\ Undecided\rangle \Rightarrow Undecided \vee (\exists rs_1\ rs_2\ m.\ r\ \#\ rs = rs_1\ @\ [Rule\ m\ Return]\ @\ rs_2 \wedge \Gamma,\gamma,p\vdash \langle rs_1,\ Undecided\rangle \Rightarrow Undecided \wedge matches\ \gamma\ m\ p)$ (**is** *?goal*)
proof(*cases matches* $\gamma\ m\ p$)
case *True*

hence $\exists rs_1\ rs_2\ m.\ r\ \#\ rs = rs_1\ @\ Rule\ m\ Return\ \#\ rs_2 \wedge \Gamma,\gamma,p\vdash \langle rs_1,\ Undecided\rangle \Rightarrow Undecided \wedge matches\ \gamma\ m\ p$

apply(*rule-tac x*=[] **in** *exI*)
apply(*rule-tac x*=*rs* **in** *exI*)
apply(*rule-tac x*=*m* **in** *exI*)
apply(*simp add*: *skip r* ‹*a* = *Return*›)
done
thus *?goal* **by** *simp*
next
case *False*
with *nomatch seq-cons False r* **have** *r-nomatch*: $\bigwedge$*rs*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided* ⟹ Γ,γ,*p*⊢ ⟨*r* # *rs*, *Undecided*⟩ ⇒ *Undecided* **by** *fast*
note *r-nomatch*′=*r-nomatch*[*simplified r* ‹*a* = *Return*›] — r unfolded
from *False not-matches-add-matchNot-simp prems* **have** Γ,γ,*p*⊢ ⟨*process-ret rs*, *Undecided*⟩ ⇒ *Undecided* **by** *fast*
with *Cons.IH* **have** *IH*: Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided* ∨ (∃ rs_1 rs_2 *m*. *rs* = rs_1 @ [*Rule m Return*] @ rs_2 ∧ Γ,γ,*p*⊢ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* ∧ *matches* γ *m p*) **.**
thus *?goal*
proof(*elim disjE*)
assume Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided*
hence Γ,γ,*p*⊢ ⟨*r* # *rs*, *Undecided*⟩ ⇒ *Undecided* **using** *r-nomatch* **by** *simp*
thus *?goal* **by** *simp*
next
assume ∃ rs_1 rs_2 *m*. *rs* = rs_1 @ [*Rule m Return*] @ rs_2 ∧ Γ,γ,*p*⊢ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* ∧ *matches* γ *m p*
from *this* **obtain** rs_1 rs_2 *m*′ **where** *rs* = rs_1 @ [*Rule m*′ *Return*] @ rs_2 **and** Γ,γ,*p*⊢ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* **and** *matches* γ *m*′ *p* **by** *blast*
hence ∃ rs_1 rs_2 *m*. *r* # *rs* = rs_1 @ [*Rule m Return*] @ rs_2 ∧ Γ,γ,*p*⊢ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* ∧ *matches* γ *m p*
apply(*rule-tac x*=*Rule m Return* # rs_1 **in** *exI*)
apply(*rule-tac x*=rs_2 **in** *exI*)
apply(*rule-tac x*=*m*′ **in** *exI*)
by(*simp add*: ‹*a* = *Return*› *False r r-nomatch*′)
thus *?goal* **by** *simp*
qed
qed
qed
qed
with *assms noreturn return* **show** *?thesis* **by** *auto*
qed

lemma *add-match-match-not-cases*:
Γ,γ,*p*⊢ ⟨*add-match* (*MatchNot m*) *rs*, *Undecided*⟩ ⇒ *Undecided* ⟹ *matches* γ *m p* ∨ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided*
by (*metis matches.simps*(*2*) *matches-add-match-simp*)

lemma *iptables-bigstep-process-ret-DecisionD*: Γ,γ,*p*⊢ ⟨*process-ret rs*, *s*⟩ ⇒ *Decision X* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *Decision X*
proof (*induction rs arbitrary*: *s*)

case (*Cons r rs*)
thus *?case*
apply(*cases r*, *rename-tac m a*)
apply(*clarify*)

apply(*case-tac a* $\neq$ *Return*)
apply(*simp add*: *process-ret-split-fst-NeqReturn*)
apply(*erule seqE-cons*)
apply(*simp add*: *seq′-cons*)

apply(*simp*)

apply(*case-tac matches* γ *m p*)
apply(*simp add*: *matches-add-match-MatchNot-simp skip*)
apply (*metis decision skipD*)

apply(*simp add*: *not-matches-add-matchNot-simp*)
by (*metis decision state.exhaust nomatch seq′-cons*)
qed *simp*

lemma *free-return-not-match*: Γ,γ,*p*$\vdash$ $\langle$[*Rule m Return*], *Undecided*$\rangle$ $\Rightarrow$ *t* $\Longrightarrow$ $\neg$ *matches* γ *m p*
using *no-free-return* **by** *fast*

3.2 Background Ruleset Updating

lemma *update-Gamma-nomatch*:
assumes $\neg$ *matches* γ *m p*
shows Γ(*chain* $\mapsto$ *Rule m a* # *rs*),γ,*p*$\vdash$ $\langle$*rs′*, *s*$\rangle$ $\Rightarrow$ *t* $\longleftrightarrow$ Γ(*chain* $\mapsto$ *rs*),γ,*p*$\vdash$ $\langle$*rs′*, *s*$\rangle$ $\Rightarrow$ *t* (**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *?l* **thus** *?r*
proof (*induction rs′ s t rule*: *iptables-bigstep-induct*)
case (*Call-return m a chain′* rs_1 *m′* rs_2)
thus *?case*
proof (*cases chain′* = *chain*)
case *True*
with *Call-return* **show** *?thesis*
apply *simp*
apply(*cases* rs_1)
using *assms* **apply** *fastforce*
apply(*rule-tac* rs_1=*list* **and** *m′*=*m′* **and** rs_2=rs_2 **in** *call-return*)
apply(*simp*)
apply(*simp*)
apply(*simp*)
apply(*simp*)

apply(*erule seqE-cons*[**where** Γ=(λa. *if a* = *chain then Some rs else*

Γ *a*)])
apply(*frule iptables-bigstep-to-undecided*[**where** Γ=(λa. *if a* = *chain then Some rs else* Γ *a*)])
apply(*simp*)
done
qed (*auto intro*: *call-return*)
next
case (*Call-result m′ a′ chain′ rs′ t′*)
have Γ(*chain* $\mapsto$ *rs*),γ,*p*$\vdash$ $\langle$[*Rule m′* (*Call chain′*)], *Undecided*$\rangle$ $\Rightarrow$ *t′*
proof (*cases chain′* = *chain*)
case *True*
with *Call-result* **have** *Rule m a* # *rs* = *rs′* (Γ(*chain* $\mapsto$ *rs*)) *chain′* = *Some rs*
by *simp*+
with *assms Call-result* **show** *?thesis*
by (*metis call-result nomatchD seqE-cons*)
next
case *False*
with *Call-result* **show** *?thesis*
by (*metis call-result fun-upd-apply*)
qed
with *Call-result* **show** *?case*
by *fast*
qed (*auto intro*: *iptables-bigstep.intros*)
next
assume *?r* **thus** *?l*
proof (*induction rs′ s t rule*: *iptables-bigstep-induct*)
case (*Call-return m′ a′ chain′* rs_1)
thus *?case*
proof (*cases chain′* = *chain*)
case *True*
with *Call-return* **show** *?thesis*
using *assms*
by (*auto intro*: *seq-cons nomatch intro*!: *call-return*[**where** rs_1 = *Rule m a* # rs_1])
qed (*auto intro*: *call-return*)
next
case (*Call-result m′ a′ chain′ rs′*)
thus *?case*
proof (*cases chain′* = *chain*)
case *True*
with *Call-result* **show** *?thesis*
using *assms* **by** (*auto intro*: *seq-cons nomatch intro*!: *call-result*)
qed (*auto intro*: *call-result*)
qed (*auto intro*: *iptables-bigstep.intros*)
qed

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
assume *?l* **thus** *?r*
proof (*induction rs' s t rule*: *iptables-bigstep-induct*)
case (*Call-return* m' a' $chain'$ rs_1 m'' rs_2)

note [*simp*] = *fun-upd-apply*[*abs-def*]

from *Call-return* **have** $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle[Rule\ m'\ (Call\ chain')],\ Undecided\rangle \Rightarrow Undecided$ (**is** *?Call-return-case*)
proof(*cases* $chain' = chain$)
case *True* **with** *Call-return* **show** *?Call-return-case*
— rs_1 cannot be empty
proof(*cases* rs_1)
case *Nil* **with** *Call-return(3)* ‹$chain' = chain$› *assms* **have** *False* **by** *simp*
thus *?Call-return-case* **by** *simp*
next
case (*Cons* r_1 rs_1s)
from *Cons Call-return* **have** $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle r_1\ \#\ rs_1s,\ Undecided\rangle \Rightarrow Undecided$ **by** *blast*
with *seqE-cons*[**where** Γ=$\Gamma(chain \mapsto rs)$] **obtain** *ti* **where**
$\Gamma(chain \mapsto rs),\gamma,p\vdash \langle[r_1],\ Undecided\rangle \Rightarrow ti$ **and** $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle rs_1s,\ ti\rangle \Rightarrow Undecided$ **by** *metis*
with *iptables-bigstep-to-undecided*[**where** Γ=$\Gamma(chain \mapsto rs)$] **have** $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle rs_1s,\ Undecided\rangle \Rightarrow Undecided$ **by** *fast*
with *Cons Call-return* ‹$chain' = chain$› **show** *?Call-return-case*
apply(*rule-tac* rs_1=rs_1s **and** m'=m'' **and** rs_2=rs_2 **in** *call-return*)
apply(*simp-all*)
done
qed
next
case *False* **with** *Call-return* **show** *?Call-return-case*
by (*auto intro*: *call-return*)
qed
thus *?case* **using** *Call-return* **by** *blast*
next
case (*Call-result* m' a' $chain'$ rs' t')
thus *?case*
proof (*cases* $chain' = chain$)
case *True*
with *Call-result* **have** $rs' = []\ @\ [Rule\ m\ a]\ @\ rs$
by *simp*
with *Call-result assms* **have** $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle[]\ @\ rs,\ Undecided\rangle \Rightarrow t'$
using *log-remove empty-empty* **by** *fast*
hence $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow t'$
by *simp*

with *Call-result True* **show** *?thesis*
by (*metis call-result fun-upd-same*)
qed (*fastforce intro*: *call-result*)
qed (*auto intro*: *iptables-bigstep.intros*)
next
have *cases-a*: $\bigwedge P.$ (a = *Log* $\Longrightarrow$ P a) $\Longrightarrow$ (a = *Empty* $\Longrightarrow$ P a) $\Longrightarrow$ P a **using** *assms* **by** *blast*
assume *?r* **thus** *?l*
proof (*induction* rs' *s t rule*: *iptables-bigstep-induct*)
case (*Call-return* m' a' $chain'$ rs_1 m'' rs_2)
from *Call-return* **have** *xx*: Γ(*chain* $\mapsto$ *Rule m a* # *rs*),γ,$p\vdash$ $\langle$*Rule m a* # rs_1, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
apply −
apply(*rule cases-a*)
apply (*auto intro*: *nomatch seq-cons intro*!: *log empty simp del*: *fun-upd-apply*)
done
with *Call-return* **show** *?case*
proof(*cases* $chain'$ = *chain*)
case *False*
with *Call-return* **have** *x*: (Γ(*chain* $\mapsto$ *Rule m a* # *rs*)) $chain'$ = *Some* (rs_1 @ *Rule* m'' *Return* # rs_2)
by(*simp*)
with *Call-return* **have** Γ(*chain* $\mapsto$ *Rule m a* # *rs*),γ,$p\vdash$ $\langle$[*Rule* m' (*Call* $chain'$)], *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
apply −
apply(*rule call-return*[**where** rs_1=rs_1 **and** m'=m'' **and** rs_2=rs_2])
apply(*simp-all add*: *x xx del*: *fun-upd-apply*)
done
thus Γ(*chain* $\mapsto$ *Rule m a* # *rs*),γ,$p\vdash$ $\langle$[*Rule* m' a'], *Undecided*$\rangle$ $\Rightarrow$ *Undecided* **using** *Call-return* **by** *simp*
next
case *True*
with *Call-return* **have** *x*: (Γ(*chain* $\mapsto$ *Rule m a* # *rs*)) $chain'$ = *Some* (*Rule m a* # rs_1 @ *Rule* m'' *Return* # rs_2)
by(*simp*)
with *Call-return* **have** Γ(*chain* $\mapsto$ *Rule m a* # *rs*),γ,$p\vdash$ $\langle$[*Rule* m' (*Call* $chain'$)], *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
apply −
apply(*rule call-return*[**where** rs_1=*Rule m a*#rs_1 **and** m'=m'' **and** rs_2=rs_2])
apply(*simp-all add*: *x xx del*: *fun-upd-apply*)
done
thus Γ(*chain* $\mapsto$ *Rule m a* # *rs*),γ,$p\vdash$ $\langle$[*Rule* m' a'], *Undecided*$\rangle$ $\Rightarrow$ *Undecided* **using** *Call-return* **by** *simp*
qed
next
case (*Call-result ma a chaina rs t*)
thus *?case*
apply (*cases chaina* = *chain*)

apply(*rule cases-a*)
apply (*auto intro*: *nomatch seq-cons intro*!: *log empty call-result*)[*2*]
by (*auto intro*!: *call-result*)[*1*]
qed (*auto intro*: *iptables-bigstep.intros*)
qed

lemma *map-update-chain-if*: (λb. *if* b = *chain then Some rs else* Γ b) = Γ(*chain* $\mapsto$ *rs*)
by *auto*

lemma *no-recursive-calls-helper*:
assumes Γ,γ,p⊢ ⟨[*Rule m* (*Call chain*)], *Undecided*⟩ ⇒ *t*
and *matches* γ *m p*
and Γ *chain* = *Some* [*Rule m* (*Call chain*)]
shows *False*
using *assms*
proof (*induction* [*Rule m* (*Call chain*)] *Undecided t rule*: *iptables-bigstep-induct*)
case *Seq*
thus *?case*
by (*metis Cons-eq-append-conv append-is-Nil-conv skipD*)
next
case (*Call-return chain′* rs_1 *m′* rs_2)
hence rs_1 @ *Rule m′ Return* # rs_2 = [*Rule m* (*Call chain′*)]
by *simp*
thus *?case*
by (*cases* rs_1) *auto*
next
case *Call-result*
thus *?case*
by *simp*
qed (*auto intro*: *iptables-bigstep.intros*)

lemma *no-recursive-calls*:
Γ(*chain* $\mapsto$ [*Rule m* (*Call chain*)]),γ,p⊢ ⟨[*Rule m* (*Call chain*)], *Undecided*⟩ ⇒ *t* $\Longrightarrow$ *matches* γ *m p* $\Longrightarrow$ *False*
by (*fastforce intro*: *no-recursive-calls-helper*)

lemma *no-recursive-calls2*:
assumes Γ(*chain* $\mapsto$ (*Rule m* (*Call chain*)) # *rs″*),γ,p⊢ ⟨(*Rule m* (*Call chain*)) # *rs′*, *Undecided*⟩ ⇒ *Undecided*
and *matches* γ *m p*
shows *False*
using *assms*
proof (*induction* (*Rule m* (*Call chain*)) # *rs′ Undecided Undecided arbitrary*: *rs′ rule*: *iptables-bigstep-induct*)
case (*Seq* rs_1 rs_2 *t*)
thus *?case*
by (*cases* rs_1) (*auto elim*: *seqE-cons simp add*: *iptables-bigstep-to-undecided*)
qed (*auto intro*: *iptables-bigstep.intros simp*: *Cons-eq-append-conv*)

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 $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle rs',\ s\rangle \Rightarrow t$
  using *assms(2)* **proof** (*induction rs′ s t rule*: *iptables-bigstep-induct*)
    case (*Call-return m a chaina* rs_1 *m′* rs_2)
    thus *?case*
      proof (*cases chaina = chain*)
        case *True*
        with *Call-return* **show** *?thesis*
          apply *simp*
          apply(*cases* rs_1)
          apply(*simp*)
          using *assms* **apply** (*metis no-free-return*)
          apply(*rule-tac* rs_1=*list* **and** $m'=m'$ **and** $rs_2=rs_2$ **in** *call-return*)
          apply(*simp*)
          apply(*simp*)
          apply(*simp*)
          apply(*simp*)
          apply(*erule seqE-cons*[**where** $\Gamma=(\lambda a.\ if\ a = chain\ then\ Some\ rs\ else\ \Gamma\ a)$])
          apply(*frule iptables-bigstep-to-undecided*[**where** $\Gamma=(\lambda a.\ if\ a = chain\ then\ Some\ rs\ else\ \Gamma\ a)$])
          apply(*simp*)
          done
      qed (*auto intro*: *call-return*)
  next
    case (*Call-result m a chaina rsa t*)
    thus *?case*
      proof (*cases chaina = chain*)
        case *True*
        with *Call-result* **show** *?thesis*
          apply(*simp*)
          apply(*cases rsa*)
          apply(*simp*)
          apply(*rule-tac rs=rs* **in** *call-result*)
          apply(*simp-all*)
          apply(*erule-tac seqE-cons*[**where** $\Gamma=(\lambda b.\ if\ b = chain\ then\ Some\ rs\ else\ \Gamma\ b)$])
          apply(*case-tac t*)
          apply(*simp*)
          apply(*frule iptables-bigstep-to-undecided*[**where** $\Gamma=(\lambda b.\ if\ b = chain\ then\ Some\ rs\ else\ \Gamma\ b)$])
          apply(*simp*)
          apply(*simp*)
          apply(*subgoal-tac ti = Undecided*)
          apply(*simp*)

using *assms*(*1*)[*simplified map-update-chain-if* [*symmetric*]] *iptables-bigstep-deterministic*
apply *fast*
done
qed (*fastforce intro*: *call-result*)
qed (*auto intro*: *iptables-bigstep.intros*)

lemma *update-gamme-remove-Undecidedpart*:
assumes Γ(*chain* $\mapsto$ *rs'*),γ,*p*$\vdash$ $\langle$*rs'*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
and Γ(*chain* $\mapsto$ *rs1*@*rs'*),γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
shows Γ(*chain* $\mapsto$*rs'*),γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
using *assms*(*2*) **proof** (*induction rs Undecided Undecided rule*: *iptables-bigstep-induct*)
case *Seq*
thus *?case*
by (*auto simp*: *iptables-bigstep-to-undecided intro*: *seq*)
next
case (*Call-return m a chaina* rs_1 *m'* rs_2)
thus *?case*
apply(*cases chaina* = *chain*)
apply(*simp*)
apply(*cases length rs1* $\leq$ *length* rs_1)
apply(*simp add*: *List.append-eq-append-conv-if*)
apply(*rule-tac* rs_1=*drop* (*length rs1*) rs_1 **and** *m'*=*m'* **and** rs_2=rs_2 **in** *call-return*)
apply(*simp-all*)[*3*]
apply(*subgoal-tac* rs_1 = (*take* (*length rs1*) rs_1) @ *drop* (*length rs1*) rs_1)
prefer *2* **apply** (*metis append-take-drop-id*)
apply(*clarify*)
apply(*subgoal-tac* Γ(*chain* $\mapsto$ *drop* (*length rs1*) rs_1 @ *Rule m' Return* # rs_2),γ,*p*$\vdash$
$\langle$(*take* (*length rs1*) rs_1) @ *drop* (*length rs1*) rs_1, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*)
prefer *2* **apply**(*auto*)[*1*]
apply(*erule-tac* rs_1=*take* (*length rs1*) rs_1 **and** rs_2=*drop* (*length rs1*) rs_1 **in** *seqE*)
apply(*simp*)
apply(*frule-tac rs*=*drop* (*length rs1*) rs_1 **in** *iptables-bigstep-to-undecided*)
apply(*simp*)

using *assms* **apply** (*auto intro*: *call-result call-return*)
done
next
case (*Call-result - - chain'* *rsa*)
thus *?case*
apply(*cases chain'* = *chain*)
apply(*simp*)
apply(*rule call-result*)
apply(*simp-all*)[*2*]
apply (*metis iptables-bigstep-to-undecided seqE*)
apply (*auto intro*: *call-result*)

done
qed (*auto intro*: *iptables-bigstep.intros*)

lemma *update-Gamma-nocall*:
assumes $\neg$ ($\exists$ *chain*. a = *Call chain*)
shows $\Gamma,\gamma,p\vdash$ $\langle$[*Rule m a*], $s\rangle \Rightarrow t \longleftrightarrow \Gamma',\gamma,p\vdash$ $\langle$[*Rule m a*], $s\rangle \Rightarrow t$
proof −
{
fix Γ Γ'
have $\Gamma,\gamma,p\vdash$ $\langle$[*Rule m a*], $s\rangle \Rightarrow t \Longrightarrow \Gamma',\gamma,p\vdash$ $\langle$[*Rule m a*], $s\rangle \Rightarrow t$
proof (*induction* [*Rule m a*] *s t rule*: *iptables-bigstep-induct*)
case *Seq*
thus *?case* **by** (*metis* (*lifting*, *no-types*) *list-app-singletonE*[**where** x = *Rule m a*] *skipD*)
next
case *Call-return* **thus** *?case* **using** *assms* **by** *metis*
next
case *Call-result* **thus** *?case* **using** *assms* **by** *metis*
qed (*auto intro*: *iptables-bigstep.intros*)
}
thus *?thesis*
by *blast*
qed

lemma *update-Gamma-call*:
assumes Γ *chain* = *Some rs* **and** Γ' *chain* = *Some rs'*
assumes $\Gamma,\gamma,p\vdash$ $\langle$*rs*, *Undecided*$\rangle \Rightarrow$ *Undecided* **and** $\Gamma',\gamma,p\vdash$ $\langle$*rs'*, *Undecided*$\rangle \Rightarrow$ *Undecided*
shows $\Gamma,\gamma,p\vdash$ $\langle$[*Rule m* (*Call chain*)], $s\rangle \Rightarrow t \longleftrightarrow \Gamma',\gamma,p\vdash$ $\langle$[*Rule m* (*Call chain*)], $s\rangle \Rightarrow t$
proof −
{
fix Γ Γ' *rs rs'*
assume *assms*:
Γ *chain* = *Some rs* Γ' *chain* = *Some rs'*
$\Gamma,\gamma,p\vdash$ $\langle$*rs*, *Undecided*$\rangle \Rightarrow$ *Undecided* $\Gamma',\gamma,p\vdash$ $\langle$*rs'*, *Undecided*$\rangle \Rightarrow$ *Undecided*
have $\Gamma,\gamma,p\vdash$ $\langle$[*Rule m* (*Call chain*)], $s\rangle \Rightarrow t \Longrightarrow \Gamma',\gamma,p\vdash$ $\langle$[*Rule m* (*Call chain*)], $s\rangle \Rightarrow t$
proof (*induction* [*Rule m* (*Call chain*)] *s t rule*: *iptables-bigstep-induct*)
case *Seq*
thus *?case* **by** (*metis* (*lifting*, *no-types*) *list-app-singletonE*[**where** x = *Rule m* (*Call chain*)] *skipD*)
next
case *Call-result*
thus *?case*
using *assms* **by** (*metis call-result iptables-bigstep-deterministic*)
qed (*auto intro*: *iptables-bigstep.intros assms*)
}
note $*$ = *this*

show *?thesis*
using $*$[*OF assms*(*1*−*4*)] $*$[*OF assms*(*2*,*1*,*4*,*3*)] **by** *blast*
qed

lemma *update-Gamma-remove-call-undecided*:
assumes Γ(*chain* $\mapsto$ *Rule m* (*Call foo*) # *rs'*),γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
and *matches* γ *m p*
shows Γ(*chain* $\mapsto$ *rs'*),γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
using *assms*
proof (*induction rs Undecided Undecided arbitrary*: *rule*: *iptables-bigstep-induct*)
case *Seq*
thus *?case*
by (*force simp*: *iptables-bigstep-to-undecided intro*: *seq'*)
next
case (*Call-return m a chaina* rs_1 *m'* rs_2)
thus *?case*
apply(*cases chaina* = *chain*)
apply(*cases* rs_1)
apply(*force intro*: *call-return*)
apply(*simp*)
apply(*erule-tac* Γ=Γ(*chain* $\mapsto$ *list* @ *Rule m' Return* # rs_2) **in** *seqE-cons*)
apply(*frule-tac* Γ=Γ(*chain* $\mapsto$ *list* @ *Rule m' Return* # rs_2) **in** *iptables-bigstep-to-undecided*)
apply(*auto intro*: *call-return*)
done
next
case (*Call-result m a chaina rsa*)
thus *?case*
apply(*cases chaina* = *chain*)
apply(*simp*)
apply (*metis call-result fun-upd-same iptables-bigstep-to-undecided seqE-cons*)
apply (*auto intro*: *call-result*)
done
qed (*auto intro*: *iptables-bigstep.intros*)

3.3 *process-ret* correctness

lemma *process-ret-add-match-dist1*: Γ,γ,*p*$\vdash$ $\langle$*process-ret* (*add-match m rs*), *s*$\rangle$ $\Rightarrow$ *t* $\Longrightarrow$ Γ,γ,*p*$\vdash$ $\langle$*add-match m* (*process-ret rs*), *s*$\rangle$ $\Rightarrow$ *t*
apply(*induction rs arbitrary*: *s t*)
apply(*simp add*: *add-match-def*)
apply(*rename-tac r rs s t*)
apply(*case-tac r*)
apply(*rename-tac m' a'*)
apply(*simp*)
apply(*case-tac a'*)
apply(*simp-all add*: *add-match-split-fst*)
apply(*erule seqE-cons*)
using *seq'* **apply**(*fastforce*)
apply(*erule seqE-cons*)

using *seq'* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq'* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq'* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq'* **apply**(*fastforce*)
defer
apply(*erule seqE-cons*)
using *seq'* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq'* **apply**(*fastforce*)
apply(*case-tac matches* γ (*MatchNot* (*MatchAnd m m'*)) *p*)
apply(*simp*)
apply (*metis decision decisionD state.exhaust matches.simps*(*1*) *matches.simps*(*2*)
matches-add-match-simp not-matches-add-match-simp)
by (*metis add-match-distrib matches.simps*(*1*) *matches.simps*(*2*) *matches-add-match-MatchNot-simp*)

lemma *process-ret-add-match-dist2*: $\Gamma,\gamma,p\vdash \langle$*add-match m* (*process-ret rs*), *s*$\rangle \Rightarrow t$
$\Longrightarrow \Gamma,\gamma,p\vdash \langle$*process-ret* (*add-match m rs*), *s*$\rangle \Rightarrow t$
apply(*induction rs arbitrary*: *s t*)
apply(*simp add*: *add-match-def*)
apply(*rename-tac r rs s t*)
apply(*case-tac r*)
apply(*rename-tac m' a'*)
apply(*simp*)
apply(*case-tac a'*)
apply(*simp-all add*: *add-match-split-fst*)
apply(*erule seqE-cons*)
using *seq'* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq'* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq'* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq'* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq'* **apply**(*fastforce*)
defer
apply(*erule seqE-cons*)
using *seq'* **apply**(*fastforce*)
apply(*erule seqE-cons*)
using *seq'* **apply**(*fastforce*)
apply(*case-tac matches* γ (*MatchNot* (*MatchAnd m m'*)) *p*)
apply(*simp*)
apply (*metis decision decisionD state.exhaust matches.simps*(*1*) *matches.simps*(*2*)
matches-add-match-simp not-matches-add-match-simp)
by (*metis add-match-distrib matches.simps*(*1*) *matches.simps*(*2*) *matches-add-match-MatchNot-simp*)

lemma *process-ret-add-match-dist*: $\Gamma,\gamma,p\vdash \langle process\text{-}ret\ (add\text{-}match\ m\ rs),\ s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle add\text{-}match\ m\ (process\text{-}ret\ rs),\ s\rangle \Rightarrow t$
by (*metis process-ret-add-match-dist1 process-ret-add-match-dist2*)

lemma *process-ret-Undecided-sound*:
assumes $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle process\text{-}ret\ (add\text{-}match\ m\ rs),\ Undecided\rangle \Rightarrow Undecided$
shows $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow Undecided$
proof (*cases matches* γ *m p*)
case *False*
thus *?thesis*
by (*metis nomatch*)
next
case *True*
note *matches = this*
show *?thesis*
using *assms* **proof** (*induction rs*)
case *Nil*
from *call-result*[*OF matches*, **where** $\Gamma=\Gamma(chain \mapsto [])$]
have $(\Gamma(chain \mapsto []))\ chain = Some\ [] \Longrightarrow \Gamma(chain \mapsto []),\gamma,p\vdash \langle[],\ Undecided\rangle \Rightarrow Undecided \Longrightarrow \Gamma(chain \mapsto []),\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow Undecided$
by *simp*
thus *?case*
by (*fastforce intro*: *skip*)
next
case (*Cons r rs*)
obtain m' a' **where** *r*: $r = Rule\ m'\ a'$ **by** (*cases r*) *blast*

with *Cons.prems* **have** *prems*: $\Gamma(chain \mapsto Rule\ m'\ a'\ \#\ rs),\gamma,p\vdash \langle process\text{-}ret\ (add\text{-}match\ m\ (Rule\ m'\ a'\ \#\ rs)),\ Undecided\rangle \Rightarrow Undecided$
by *fast*
hence *prems-simplified*: $\Gamma(chain \mapsto Rule\ m'\ a'\ \#\ rs),\gamma,p\vdash \langle process\text{-}ret\ (Rule\ m'\ a'\ \#\ rs),\ Undecided\rangle \Rightarrow Undecided$
using *matches* **by** (*metis matches-add-match-simp process-ret-add-match-dist*)

have $\Gamma(chain \mapsto Rule\ m'\ a'\ \#\ rs),\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow Undecided$
proof (*cases* $a' = Return$)
case *True*
note $a' = this$
have $\Gamma(chain \mapsto Rule\ m'\ Return\ \#\ rs),\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow Undecided$
proof (*cases matches* γ m' *p*)
case *True*
with *matches* **show** *?thesis*
by (*fastforce intro*: *call-return skip*)

next
case *False*
note *matches'* = *this*
hence Γ(*chain* $\mapsto$ *rs*),γ,*p*$\vdash$ $\langle$*process-ret* (*Rule m' a'* # *rs*), *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
by (*metis prems-simplified update-Gamma-nomatch*)
with *a'* **have** Γ(*chain* $\mapsto$ *rs*),γ,*p*$\vdash$ $\langle$*add-match* (*MatchNot m'*) (*process-ret rs*), *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
by *simp*
with *matches matches'* **have** Γ(*chain* $\mapsto$ *rs*),γ,*p*$\vdash$ $\langle$*add-match m* (*process-ret rs*), *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
by (*simp add*: *matches-add-match-simp not-matches-add-matchNot-simp*)
with *matches' Cons.IH* **show** *?thesis*
by (*fastforce simp*: *update-Gamma-nomatch process-ret-add-match-dist*)
qed
with *a'* **show** *?thesis*
by *simp*
next
case *False*
note *a'* = *this*
with *prems-simplified* **have** Γ(*chain* $\mapsto$ *Rule m' a'* # *rs*),γ,*p*$\vdash$ $\langle$*Rule m' a'* # *process-ret rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
by (*simp add*: *process-ret-split-fst-NeqReturn*)
hence *step*: Γ(*chain* $\mapsto$ *Rule m' a'* # *rs*),γ,*p*$\vdash$ $\langle$[*Rule m' a'*], *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
and *IH-pre*: Γ(*chain* $\mapsto$ *Rule m' a'* # *rs*),γ,*p*$\vdash$ $\langle$*process-ret rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
by (*metis seqE-cons iptables-bigstep-to-undecided*)+

from *step* **have** Γ(*chain* $\mapsto$ *rs*),γ,*p*$\vdash$ $\langle$*process-ret rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
proof (*cases rule*: *Rule-UndecidedE*)
case *log* **thus** *?thesis*
using *IH-pre* **by** (*metis empty iptables-bigstep.log update-Gamma-nochange1 update-Gamma-nomatch*)
next
case *call* **thus** *?thesis*
using *IH-pre* **by** (*metis update-Gamma-remove-call-undecided*)
next
case *nomatch* **thus** *?thesis*
using *IH-pre* **by** (*metis update-Gamma-nomatch*)
qed

hence Γ(*chain* $\mapsto$ *rs*),γ,*p*$\vdash$ $\langle$*process-ret* (*add-match m rs*), *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
by (*metis matches matches-add-match-simp process-ret-add-match-dist*)
with *Cons.IH* **have** *IH*: Γ(*chain* $\mapsto$ *rs*),γ,*p*$\vdash$ $\langle$[*Rule m* (*Call chain*)], *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
by *fast*

from *step* **show** *?thesis*
proof (*cases rule*: *Rule-UndecidedE*)
case *log* **thus** *?thesis* **using** *IH*
by (*simp add*: *update-Gamma-log-empty*)
next
case *nomatch*
thus *?thesis*
using *IH* **by** (*metis update-Gamma-nomatch*)
next
case (*call c*)
let $?\Gamma' = \Gamma(\mathit{chain} \mapsto \mathit{Rule}\ m'\ a'\ \#\ \mathit{rs})$
from *IH-pre* **show** *?thesis*
proof (*cases rule*: *iptables-bigstep-process-ret-cases3*)
case *noreturn*
with *call* **have** $?\Gamma',\gamma,p\vdash \langle \mathit{Rule}\ m'\ (\mathit{Call}\ c)\ \#\ \mathit{rs},\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}$
by (*metis step seq-cons*)
from *call* **have** $?\Gamma'\ \mathit{chain} = \mathit{Some}\ (\mathit{Rule}\ m'\ (\mathit{Call}\ c)\ \#\ \mathit{rs})$
by *simp*
from *matches* **show** *?thesis*
by (*rule call-result*) *fact+*
next
case (*return* rs_1 rs_2 *new-m'*)
with *call* **have** $?\Gamma'\ \mathit{chain} = \mathit{Some}\ ((\mathit{Rule}\ m'\ (\mathit{Call}\ c)\ \#\ rs_1)\ @\ [\mathit{Rule}\ \mathit{new}\text{-}m'\ \mathit{Return}]\ @\ rs_2)$
by *simp*
from *call return step* **have** $?\Gamma',\gamma,p\vdash \langle \mathit{Rule}\ m'\ (\mathit{Call}\ c)\ \#\ rs_1,\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}$
using *IH-pre* **by** (*auto intro*: *seq-cons*)
from *matches* **show** *?thesis*
by (*rule call-return*) *fact+*
qed
qed
qed
thus *?case*
by (*metis r*)
qed
qed

lemma *process-ret-Decision-sound*:
assumes $\Gamma(\mathit{chain} \mapsto \mathit{rs}),\gamma,p\vdash \langle \mathit{process\text{-}ret}\ (\mathit{add\text{-}match}\ m\ \mathit{rs}),\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ X$
shows $\Gamma(\mathit{chain} \mapsto \mathit{rs}),\gamma,p\vdash \langle [\mathit{Rule}\ m\ (\mathit{Call}\ \mathit{chain})],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ X$
proof (*cases matches* γ *m p*)
case *False*
thus *?thesis* **by** (*metis assms state.distinct(1) not-matches-add-match-simp process-ret-add-match-dist1 skipD*)
next

case *True*
note *matches* = *this*
show *?thesis*
using *assms* **proof** (*induction rs*)
case *Nil*
hence *False* **by** (*metis add-match-split append-self-conv state.distinct*(*1*) *process-ret.simps*(*1*) *skipD*)
thus *?case* **by** *simp*
next
case (*Cons r rs*)
obtain m' a' **where** *r*: $r = Rule\ m'\ a'$ **by** (*cases r*) *blast*

with *Cons.prems* **have** *prems*: $\Gamma(chain \mapsto Rule\ m'\ a' \# rs),\gamma,p\vdash \langle process\text{-}ret\ (add\text{-}match\ m\ (Rule\ m'\ a' \# rs)),\ Undecided\rangle \Rightarrow Decision\ X$
by *fast*
hence *prems-simplified*: $\Gamma(chain \mapsto Rule\ m'\ a' \# rs),\gamma,p\vdash \langle process\text{-}ret\ (Rule\ m'\ a' \# rs),\ Undecided\rangle \Rightarrow Decision\ X$
using *matches* **by** (*metis matches-add-match-simp process-ret-add-match-dist*)

have $\Gamma(chain \mapsto Rule\ m'\ a' \# rs),\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow Decision\ X$
proof (*cases* $a' = Return$)
case *True*
note $a' = this$
have $\Gamma(chain \mapsto Rule\ m'\ Return \# rs),\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow Decision\ X$
proof (*cases matches* γ m' *p*)
case *True*
with *matches prems-simplified* a' **show** *?thesis*
by (*auto simp*: *not-matches-add-match-simp dest*: *skipD*)
next
case *False*
note $matches' = this$
with *prems-simplified* **have** $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle process\text{-}ret\ (Rule\ m'\ a' \# rs),\ Undecided\rangle \Rightarrow Decision\ X$
by (*metis update-Gamma-nomatch*)
with a' *matches* $matches'$ **have** $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle add\text{-}match\ m\ (process\text{-}ret\ rs),\ Undecided\rangle \Rightarrow Decision\ X$
by (*simp add*: *matches-add-match-simp not-matches-add-matchNot-simp*)
with *matches* $matches'$ *Cons.IH* **show** *?thesis*
by (*fastforce simp*: *update-Gamma-nomatch process-ret-add-match-dist matches-add-match-simp not-matches-add-matchNot-simp*)
qed
with a' **show** *?thesis*
by *simp*
next
case *False*
with *prems-simplified* **obtain** *ti*
where *step*: $\Gamma(chain \mapsto Rule\ m'\ a' \# rs),\gamma,p\vdash \langle[Rule\ m'\ a'],\ Undecided\rangle$

$\Rightarrow$ *ti*
      and *IH-pre*: $\Gamma(chain \mapsto Rule\ m'\ a'\ \#\ rs),\gamma,p\vdash \langle process\text{-}ret\ rs,\ ti\rangle \Rightarrow Decision\ X$
      by (*auto simp*: *process-ret-split-fst-NeqReturn elim*: *seqE-cons*)

    hence $\Gamma(chain \mapsto Rule\ m'\ a'\ \#\ rs),\gamma,p\vdash \langle rs,\ ti\rangle \Rightarrow Decision\ X$
      by (*metis iptables-bigstep-process-ret-DecisionD*)

    thus *?thesis*
      using *matches step* **by** (*force intro*: *call-result seq'-cons*)
    qed
   thus *?case*
    by (*metis r*)
  qed
 qed

lemma *process-ret-result-empty*: $[] = process\text{-}ret\ rs \Longrightarrow \forall r \in set\ rs.\ get\text{-}action\ r = Return$
 proof (*induction rs*)
  case (*Cons r rs*)
  thus *?case*
   apply(*simp*)
   apply(*case-tac r*)
   apply(*rename-tac m a*)
   apply(*case-tac a*)
   apply(*simp-all add*: *add-match-def*)
   done
 qed *simp*

lemma *all-return-subchain*:
 assumes *a1*: $\Gamma\ chain = Some\ rs$
 and *a2*: $matches\ \gamma\ m\ p$
 and *a3*: $\forall r{\in}set\ rs.\ get\text{-}action\ r = Return$
 shows $\Gamma,\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow Undecided$
 proof (*cases* $\exists r \in set\ rs.\ matches\ \gamma\ (get\text{-}match\ r)\ p$)
  case *True*
  hence $(\exists rs1\ r\ rs2.\ rs = rs1\ @\ r\ \#\ rs2 \wedge matches\ \gamma\ (get\text{-}match\ r)\ p \wedge (\forall r'{\in}set\ rs1.\ \neg\ matches\ \gamma\ (get\text{-}match\ r')\ p))$
   by (*subst split-list-first-prop-iff* [*symmetric*])
  then obtain *rs1 r rs2*
    where $*$: $rs = rs1\ @\ r\ \#\ rs2$ $matches\ \gamma\ (get\text{-}match\ r)\ p$ $\forall r'{\in}set\ rs1.\ \neg\ matches\ \gamma\ (get\text{-}match\ r')\ p$
   by *auto*

  with *a3* **obtain** m' **where** $r = Rule\ m'\ Return$
   by (*cases r*) *simp*
  with $*$ *assms* **show** *?thesis*
   by (*fastforce intro*: *call-return nomatch'*)
 next

case *False*
hence $\Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Undecided$
by (*blast intro*: *nomatch'*)
with *a1 a2* **show** *?thesis*
by (*metis call-result*)
qed

lemma *process-ret-sound'*:
assumes $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle process\text{-}ret\ (add\text{-}match\ m\ rs),\ Undecided\rangle \Rightarrow t$
shows $\Gamma(chain \mapsto rs),\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow t$
using *assms* **by** (*metis state.exhaust process-ret-Undecided-sound process-ret-Decision-sound*)

lemma *get-action-case-simp*: $get\text{-}action\ (case\ r\ of\ Rule\ m'\ x \Rightarrow Rule\ (MatchAnd\ m\ m')\ x) = get\text{-}action\ r$
by (*metis rule.case-eq-if rule.sel*(*2*))

We call a ruleset wf iff all Calls are into actually existing chains.

definition *wf-chain* :: $'a\ ruleset \Rightarrow 'a\ rule\ list \Rightarrow bool$ **where**
$wf\text{-}chain\ \Gamma\ rs \equiv (\forall r \in set\ rs.\ \forall\ chain.\ get\text{-}action\ r = Call\ chain \longrightarrow \Gamma\ chain \neq None)$
lemma *wf-chain-append*: $wf\text{-}chain\ \Gamma\ (rs1@rs2) \longleftrightarrow wf\text{-}chain\ \Gamma\ rs1 \wedge wf\text{-}chain\ \Gamma\ rs2$
by(*simp add*: *wf-chain-def*, *blast*)
lemma *wf-chain-process-ret*: $wf\text{-}chain\ \Gamma\ rs \Longrightarrow wf\text{-}chain\ \Gamma\ (process\text{-}ret\ rs)$
apply(*induction rs*)
apply(*simp add*: *wf-chain-def add-match-def*)
apply(*case-tac a*)
apply(*case-tac* $x2 \neq Return$)
apply(*simp add*: *process-ret-split-fst-NeqReturn*)
using *wf-chain-append* **apply** (*metis Cons-eq-appendI append-Nil*)
apply(*simp add*: *process-ret-split-fst-Return*)
apply(*simp add*: *wf-chain-def add-match-def get-action-case-simp*)
done
lemma *wf-chain-add-match*: $wf\text{-}chain\ \Gamma\ rs \Longrightarrow wf\text{-}chain\ \Gamma\ (add\text{-}match\ m\ rs)$
by(*induction rs*) (*simp-all add*: *wf-chain-def add-match-def get-action-case-simp*)

3.4 Soundness

theorem *unfolding-sound*: $wf\text{-}chain\ \Gamma\ rs \Longrightarrow \Gamma,\gamma,p\vdash \langle process\text{-}call\ \Gamma\ rs,\ s\rangle \Rightarrow t \Longrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
proof (*induction rs arbitrary*: *s t*)
case (*Cons r rs*)
thus *?case*
apply −
apply(*subst*(*asm*) *process-call-split-fst*)
apply(*erule seqE*)

unfolding *wf-chain-def*

apply(*case-tac r*, *rename-tac m a*)
apply(*case-tac a*)
apply(*simp-all add*: *seq′-cons*)

apply(*case-tac s*)
defer
apply (*metis decision decisionD*)
apply(*case-tac matches* γ *m p*)
defer
apply(*simp add*: *not-matches-add-match-simp*)
apply(*drule skipD*, *simp*)
apply (*metis nomatch seq-cons*)
apply(*clarify*)
apply(*simp add*: *matches-add-match-simp*)
apply(*rule-tac t*=*ti* **in** *seq-cons*)
apply(*simp-all*)

using *process-ret-sound′*
by (*metis fun-upd-triv matches-add-match-simp process-ret-add-match-dist*)
qed *simp*

corollary *unfolding-sound-complete*: *wf-chain* Γ *rs* $\Longrightarrow$ $\Gamma,\gamma,p\vdash$ $\langle$*process-call* Γ *rs*, *s*$\rangle$ $\Rightarrow$ *t* $\longleftrightarrow$ $\Gamma,\gamma,p\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow$ *t*
by (*metis unfolding-complete unfolding-sound*)

corollary *unfolding-n-sound-complete*: $\forall$ *rsg* $\in$ *ran* Γ $\cup$ {*rs*}. *wf-chain* Γ *rsg* $\Longrightarrow$ $\Gamma,\gamma,p\vdash$ $\langle$((*process-call* Γ) ^^*n*) *rs*, *s*$\rangle$ $\Rightarrow$ *t* $\longleftrightarrow$ $\Gamma,\gamma,p\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow$ *t*
proof(*induction n arbitrary*: *rs*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)
from *Suc* **have** $\Gamma,\gamma,p\vdash$ $\langle$(*process-call* Γ ^^ *n*) *rs*, *s*$\rangle$ $\Rightarrow$ *t* = $\Gamma,\gamma,p\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow$ *t* **by** *blast*
from *Suc.prems* **have** $\forall$ *a*$\in$*ran* Γ $\cup$ {*process-call* Γ *rs*}. *wf-chain* Γ *a*
proof(*induction rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case(*Cons r rs*)
from *Cons.prems* **have** $\forall$ *a*$\in$*ran* Γ. *wf-chain* Γ *a* **by** *blast*
from *Cons.prems* **have** *wf-chain* Γ [*r*]
apply(*simp*)
apply(*clarify*)
apply(*simp add*: *wf-chain-def*)
done
from *Cons.prems* **have** *wf-chain* Γ *rs*
apply(*simp*)
apply(*clarify*)
apply(*simp add*: *wf-chain-def*)
done

from *this Cons.prems Cons.IH* **have** *wf-chain* Γ (*process-call* Γ *rs*) **by** *blast*
from *this* ‹*wf-chain* Γ [*r*]›**have** *wf-chain* Γ (*r* # (*process-call* Γ *rs*)) **by**(*simp add*: *wf-chain-def*)
from *this Cons.prems* **have** *wf-chain* Γ (*process-call* Γ (*r*#*rs*))
apply(*cases r*)
apply(*rename-tac m a*, *clarify*)
apply(*case-tac a*)
apply(*simp-all*)
apply(*simp add*: *wf-chain-append*)
apply(*clarify*)
apply(*simp add*: ‹*wf-chain* Γ (*process-call* Γ *rs*)›)
apply(*rule wf-chain-add-match*)
apply(*rule wf-chain-process-ret*)
apply(*simp add*: *wf-chain-def*)
apply(*clarify*)
by (*metis ranI option.sel*)
from *this* ‹∀ *a*∈*ran* Γ. *wf-chain* Γ *a*› **show** *?case* **by** *simp*
qed
from *this Suc.IH*[*of* ((*process-call* Γ) *rs*)] **have**
Γ,γ,*p*⊢ ⟨(*process-call* Γ ^^ *n*) (*process-call* Γ *rs*), *s*⟩ ⇒ *t* = Γ,γ,*p*⊢ ⟨*process-call* Γ *rs*, *s*⟩ ⇒ *t*
by *simp*
from *this* **show** *?case*
by (*simp*, *metis Suc.prems Un-commute funpow-swap1 insertI1 insert-is-Un unfolding-sound-complete*)
qed

```
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
```

end
theory *Ternary*
imports *Main*
begin

4 Ternary Logic

Kleene logic

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

fun *ternary-to-bool* :: *ternaryvalue* ⇒ *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* ⇒ *ternaryvalue* **where**
bool-to-ternary True = *TernaryTrue* |
bool-to-ternary False = *TernaryFalse*

lemma *the* ∘ *ternary-to-bool* ∘ *bool-to-ternary* = *id*
by(*simp add*: *fun-eq-iff*, *clarify*, *case-tac x*, *simp-all*)
lemma *ternary-to-bool-bool-to-ternary*: *ternary-to-bool* (*bool-to-ternary X*) = *Some X*
by(*cases X*, *simp-all*)
lemma *ternary-to-bool-None*: *ternary-to-bool t* = *None* ⟷ *t* = *TernaryUnknown*
by(*cases t*, *simp-all*)
lemma *ternary-to-bool-SomeE*: *ternary-to-bool t* = *Some X* ⟹
(*t* = *TernaryTrue* ⟹ *X* = *True* ⟹ *P*) ⟹ (*t* = *TernaryFalse* ⟹ *X* = *False* ⟹ *P*) ⟹ *P*
by (*metis option.distinct*(*1*) *option.inject ternary-to-bool.elims*)
lemma *ternary-to-bool-Some*: *ternary-to-bool t* = *Some X* ⟷ (*t* = *TernaryTrue* ∧ *X* = *True*) ∨ (*t* = *TernaryFalse* ∧ *X* = *False*)
by(*cases t*, *simp-all*)
lemma *bool-to-ternary-Unknown*: *bool-to-ternary t* = *TernaryUnknown* ⟷ *False*
by(*cases t*, *simp-all*)

fun *eval-ternary-And* :: *ternaryvalue* ⇒ *ternaryvalue* ⇒ *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*
by (*cases t1 t2 rule*: *ternaryvalue.exhaust*[*case-product ternaryvalue.exhaust*]) *auto*

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*
apply(*case-tac a*)
apply(*case-tac* [!] *b*)
apply(*simp-all*)
done

lemma *eval-ternary-Not-UnknownD*: *eval-ternary-Not t = TernaryUnknown* ⟹ *t = TernaryUnknown*
by (*cases t*) *auto*

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*)
by (*cases a b rule*: *ternaryvalue.exhaust*[*case-product ternaryvalue.exhaust*],*auto*)+

lemma *eval-ternary-idempotence-Not*: *eval-ternary-Not* (*eval-ternary-Not a*) = *a*
by (*cases a*) *simp-all*

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
by(*case-tac* [!] *x*)(*simp-all*)

context
begin
private lemma *bool-to-ternary-simp1*: *bool-to-ternary X* = *TernaryTrue* $\longleftrightarrow$ *X*
by (*metis bool-to-ternary.elims ternaryvalue.distinct*(*1*))
private lemma *bool-to-ternary-simp2*: *bool-to-ternary Y* = *TernaryFalse* $\longleftrightarrow$ $\neg$ *Y*
by (*metis bool-to-ternary.elims ternaryvalue.distinct*(*1*))
private lemma *bool-to-ternary-simp3*: *eval-ternary-Not* (*bool-to-ternary X*) = *TernaryTrue* $\longleftrightarrow$ $\neg$ *X*
by (*metis* (*full-types*) *bool-to-ternary-simp2 eval-ternary-Not.simps*(*1*) *eval-ternary-idempotence-Not*)
private lemma *bool-to-ternary-simp4*: *eval-ternary-Not* (*bool-to-ternary X*) = *TernaryFalse* $\longleftrightarrow$ *X*
by (*metis bool-to-ternary-simp1 eval-ternary-Not.simps*(*1*) *eval-ternary-idempotence-Not*)
private lemma *bool-to-ternary-simp5*: $\neg$ (*eval-ternary-Not* (*bool-to-ternary X*) = *TernaryUnknown*)
by (*metis bool-to-ternary-Unknown eval-ternary-Not-UnknownD*)

private lemma *bool-to-ternary-simp6*: *bool-to-ternary X* $\neq$ *TernaryUnknown*
by (*metis* (*full-types*) *bool-to-ternary.simps*(*1*) *bool-to-ternary.simps*(*2*) *ternaryvalue.distinct*(*3*) *ternaryvalue.distinct*(*5*))

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*)
by(*cases X*)(*simp-all*)

private lemma *bool-to-ternary-pullup2*: *eval-ternary-And* (*bool-to-ternary X1*) (*bool-to-ternary X2*) = *bool-to-ternary* (*X1* $\wedge$ *X2*)
by (*metis bool-to-ternary-simps*(*1*) *bool-to-ternary-simps*(*2*) *eval-ternary-simps-simple*(*2*) *eval-ternary-simps-simple*(*4*))

private lemma *bool-to-ternary-pullup3*: *eval-ternary-Imp* (*bool-to-ternary X1*) (*bool-to-ternary X2*) = *bool-to-ternary* (*X1* $\longrightarrow$ *X2*)
by (*metis bool-to-ternary-simps*(*1*) *bool-to-ternary-simps*(*2*) *eval-ternary-Imp.simps*(*1*)

eval-ternary-Imp.simps(*2*) *eval-ternary-Imp.simps*(*3*) *eval-ternary-Imp.simps*(*4*))

private lemma *bool-to-ternary-pullup4*: *eval-ternary-Or* (*bool-to-ternary X1*) (*bool-to-ternary X2*) = *bool-to-ternary* (*X1* $\vee$ *X2*)
by (*metis* (*full-types*) *bool-to-ternary.simps*(*1*) *bool-to-ternary.simps*(*2*) *eval-ternary-Or.simps*(*1*)

eval-ternary-Or.simps(2) eval-ternary-Or.simps(3) eval-ternary-Or.simps(4))

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* ⇒ *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*))
by (*simp add*: *eval-ternary-DeMorgan*)

lemma *ternary-ternary-eval-idempotence-Not*: *ternary-ternary-eval* (*TernaryNot* (*TernaryNot a*)) = *ternary-ternary-eval a*
by (*simp add*: *eval-ternary-idempotence-Not*)

lemma *ternary-ternary-eval-TernaryAnd-comm*: *ternary-ternary-eval* (*TernaryAnd t1 t2*) = *ternary-ternary-eval* (*TernaryAnd t2 t1*)
by (*simp add*: *eval-ternary-And-comm*)

lemma *eval-ternary-Not* (*ternary-ternary-eval t*) = (*ternary-ternary-eval* (*TernaryNot t*)) **by** *simp*

context
begin
private lemma *eval-ternary-simps-2*:
eval-ternary-And (*bool-to-ternary P*) *T* = *TernaryTrue* ⟷ *P* ∧ *T* = *TernaryTrue*
eval-ternary-And T (*bool-to-ternary P*) = *TernaryTrue* ⟷ *P* ∧ *T* = *TernaryTrue*
apply(*case-tac* [!] *P*)
apply(*simp-all add*: *eval-ternary-simps-simple*)
done

private lemma *eval-ternary-simps-3*:
eval-ternary-And (*ternary-ternary-eval x*) *T* = *TernaryTrue* ⟷ (*ternary-ternary-eval x* = *TernaryTrue*) ∧ (*T* = *TernaryTrue*)
eval-ternary-And T (*ternary-ternary-eval x*) = *TernaryTrue* ⟷ (*ternary-ternary-eval*

x = *TernaryTrue*) ∧ (*T* = *TernaryTrue*)
apply(*case-tac* [!] *T*)
apply(*simp-all add*: *eval-ternary-simps-simple*)
apply(*case-tac* [!] (*ternary-ternary-eval x*))
apply(*simp-all*)
done

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

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

4.1 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* ⇒ *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* ⇒ *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(*induction t rule*: *NNF-ternary.induct*)
qed(*simp-all add*: *eval-ternary-DeMorgan eval-ternary-idempotence-Not*)

lemma *NNF-ternary-NegationNormalForm*: *NegationNormalForm* (*NNF-ternary t*)

proof(*induction t rule*: *NNF-ternary.induct*)
qed(*auto simp add*: *eval-ternary-DeMorgan eval-ternary-idempotence-Not intro*: *NegationNormalForm.intros*)

end
theory *Matching-Ternary*
imports ../*Common*/*Ternary* ../*Firewall-Common*
begin

5 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 *ternaryformula*s we construct never have Or expressions.

private fun *ternary-has-or* :: *ternaryformula* $\Rightarrow$ *bool* **where**
ternary-has-or (*TernaryOr* - -) $\longleftrightarrow$ *True* |
ternary-has-or (*TernaryAnd t1 t2*) $\longleftrightarrow$ *ternary-has-or t1* $\vee$ *ternary-has-or t2* |
ternary-has-or (*TernaryNot t*) $\longleftrightarrow$ *ternary-has-or t* |
ternary-has-or (*TernaryValue* -) $\longleftrightarrow$ *False*
private lemma *map-match-tac--does-not-use-TernaryOr*: $\neg$ (*ternary-has-or* (*map-match-tac* β *p m*))
by(*induction m*, *simp-all*)

declare *ternary-has-or.simps*[*simp del*]
end

fun *ternary-to-bool-unknown-match-tac* :: *'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*))
unfolding *matches-def* **by** *simp*

lemma *matches-case*: *matches* γ *m a p* ⟷ (*case ternary-eval* (*map-match-tac* (*fst* γ) *p m*) *of None* ⇒ (*snd* γ) *a p* | *Some b* ⇒ *b*)
unfolding *matches-def ternary-eval-def*
by (*cases* (*ternary-ternary-eval* (*map-match-tac* (*fst* γ) *p m*))) *auto*

lemma *matches-case-tuple*: *matches* (β, α) *m a p* ⟷ (*case ternary-eval* (*map-match-tac* β *p m*) *of None* ⇒ α *a p* | *Some b* ⇒ *b*)
by (*auto simp*: *matches-case split*: *option.splits*)

lemma *matches-case-ternaryvalue-tuple*: *matches* (β, α) *m a p* ⟷ (*case ternary-ternary-eval* (*map-match-tac* β *p m*) *of*
TernaryUnknown ⇒ α *a p* |
TernaryTrue ⇒ *True* |
TernaryFalse ⇒ *False*)
by(*simp split*: *option.split ternaryvalue.split add*: *matches-case ternary-to-bool-None ternary-eval-def*)

lemma *matches-casesE*:
$matches\ (\beta, \alpha)\ m\ a\ p \Longrightarrow$
$(ternary\text{-}ternary\text{-}eval\ (map\text{-}match\text{-}tac\ \beta\ p\ m) = TernaryUnknown \Longrightarrow \alpha\ a\ p \Longrightarrow P) \Longrightarrow$
$(ternary\text{-}ternary\text{-}eval\ (map\text{-}match\text{-}tac\ \beta\ p\ m) = TernaryTrue \Longrightarrow P)$
$\Longrightarrow P$
proof(*induction m*)
qed(*auto split*: *option.split-asm simp*: *matches-case-tuple ternary-eval-def ternary-to-bool-bool-to-ternary elim*: *ternary-to-bool.elims*)

Example: ¬ *Unknown* is as good as *Unknown*

lemma $[\![\ ternary\text{-}ternary\text{-}eval\ (map\text{-}match\text{-}tac\ \beta\ p\ expr) = TernaryUnknown\ ]\!] \Longrightarrow matches\ (\beta, \alpha)\ expr\ a\ p \longleftrightarrow matches\ (\beta, \alpha)\ (MatchNot\ expr)\ a\ p$
by(*simp add*: *matches-case-ternaryvalue-tuple*)

lemma *bunch-of-lemmata-about-matches*:
$matches\ \gamma\ (MatchAnd\ m1\ m2)\ a\ p \longleftrightarrow matches\ \gamma\ m1\ a\ p \wedge matches\ \gamma\ m2\ a\ p$
$matches\ \gamma\ MatchAny\ a\ p$
$matches\ \gamma\ (MatchNot\ MatchAny)\ a\ p \longleftrightarrow False$
$matches\ (\beta, \alpha)\ (Match\ expr)\ a\ p = (case\ ternary\text{-}to\text{-}bool\ (\beta\ expr\ p)\ of\ Some\ r \Rightarrow r\ |\ None \Rightarrow (\alpha\ a\ p))$
$matches\ (\beta, \alpha)\ (Match\ expr)\ a\ p = (case\ (\beta\ expr\ p)\ of\ TernaryTrue \Rightarrow True\ |\ TernaryFalse \Rightarrow False\ |\ TernaryUnknown \Rightarrow (\alpha\ a\ p))$
$matches\ \gamma\ (MatchNot\ (MatchNot\ m))\ a\ p \longleftrightarrow matches\ \gamma\ m\ a\ p$
proof(*case-tac* [!] γ)
qed (*simp-all split*: *ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)

lemma *matches-DeMorgan*: $matches\ \gamma\ (MatchNot\ (MatchAnd\ m1\ m2))\ a\ p \longleftrightarrow (matches\ \gamma\ (MatchNot\ m1)\ a\ p) \vee (matches\ \gamma\ (MatchNot\ m2)\ a\ p)$
by (*cases* γ) (*simp split*: *ternaryvalue.split add*: *matches-case-ternaryvalue-tuple eval-ternary-DeMorgan*)

5.1 Ternary Matcher Algebra

lemma *matches-and-comm*: $matches\ \gamma\ (MatchAnd\ m\ m')\ a\ p \longleftrightarrow matches\ \gamma\ (MatchAnd\ m'\ m)\ a\ p$
apply(*cases* γ, *rename-tac* $\beta\ \alpha$, *clarify*)
by(*simp split*: *ternaryvalue.split add*: *matches-case-ternaryvalue-tuple eval-ternary-And-comm*)

lemma *matches-not-idem*: $matches\ \gamma\ (MatchNot\ (MatchNot\ m))\ a\ p \longleftrightarrow matches\ \gamma\ m\ a\ p$
by (*metis bunch-of-lemmata-about-matches*(*6*))

lemma $(TernaryNot\ (map\text{-}match\text{-}tac\ \beta\ p\ (m))) = (map\text{-}match\text{-}tac\ \beta\ p\ (MatchNot$

m))
by (*metis map-match-tac.simps*(*2*))

context
begin
private lemma *matches-simp1*: *matches* γ *m a p* $\Longrightarrow$ *matches* γ (*MatchAnd m m'*) *a p* $\longleftrightarrow$ *matches* γ *m' a p*
apply(*cases* γ, *rename-tac* β α, *clarify*)
apply(*simp split*: *ternaryvalue.split-asm ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)
done

private lemma *matches-simp11*: *matches* γ *m a p* $\Longrightarrow$ *matches* γ (*MatchAnd m' m*) *a p* $\longleftrightarrow$ *matches* γ *m' a p*
by(*simp-all add*: *matches-and-comm matches-simp1*)

private lemma *matches-simp2*: *matches* γ (*MatchAnd m m'*) *a p* $\Longrightarrow$ $\neg$ *matches* γ *m a p* $\Longrightarrow$ *False*
by (*metis bunch-of-lemmata-about-matches*(*1*))
private lemma *matches-simp22*: *matches* γ (*MatchAnd m m'*) *a p* $\Longrightarrow$ $\neg$ *matches* γ *m' a p* $\Longrightarrow$ *False*
by (*metis bunch-of-lemmata-about-matches*(*1*))

private lemma *matches-simp3*: *matches* γ (*MatchNot m*) *a p* $\Longrightarrow$ *matches* γ *m a p* $\Longrightarrow$ (*snd* γ) *a p*
apply(*cases* γ, *rename-tac* β α, *clarify*)
apply(*simp split*: *ternaryvalue.split-asm ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)
done
private lemma *matches* γ (*MatchNot m*) *a p* $\Longrightarrow$ *matches* γ *m a p* $\Longrightarrow$ (*ternary-eval* (*map-match-tac* (*fst* γ) *p m*)) = *None*
apply(*cases* γ, *rename-tac* β α, *clarify*)
apply(*simp split*: *ternaryvalue.split-asm ternaryvalue.split add*: *matches-case-ternaryvalue-tuple ternary-eval-def*)
done

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*) $\Longrightarrow$ *matches* (β,α) (*f* (β,α) *a m*) *a p* $\longleftrightarrow$ *matches* (β,α) *m a p*
by(*simp split*: *ternaryvalue.split-asm ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)

lemma *matches-iff-apply-f*: *ternary-ternary-eval* (*map-match-tac* β *p* (*f m*)) = *ternary-ternary-eval* (*map-match-tac* β *p m*) $\Longrightarrow$ *matches* (β,α) (*f m*) *a p* $\longleftrightarrow$ *matches* (β,α) *m a p*
by(*simp split*: *ternaryvalue.split-asm ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)

Optimize away MatchAny matches

fun *opt-MatchAny-match-expr* :: *'a match-expr* ⇒ *'a match-expr* **where**
opt-MatchAny-match-expr MatchAny = *MatchAny* |
opt-MatchAny-match-expr (*Match a*) = (*Match a*) |
opt-MatchAny-match-expr (*MatchNot* (*MatchNot m*)) = (*opt-MatchAny-match-expr m*) |
opt-MatchAny-match-expr (*MatchNot m*) = *MatchNot* (*opt-MatchAny-match-expr m*) |
opt-MatchAny-match-expr (*MatchAnd MatchAny MatchAny*) = *MatchAny* |
opt-MatchAny-match-expr (*MatchAnd MatchAny m*) = (*opt-MatchAny-match-expr m*) |
opt-MatchAny-match-expr (*MatchAnd m MatchAny*) = (*opt-MatchAny-match-expr m*) |
opt-MatchAny-match-expr (*MatchAnd* - (*MatchNot MatchAny*)) = (*MatchNot MatchAny*) |
opt-MatchAny-match-expr (*MatchAnd* (*MatchNot MatchAny*) -) = (*MatchNot MatchAny*) |
opt-MatchAny-match-expr (*MatchAnd m1 m2*) = *MatchAnd* (*opt-MatchAny-match-expr m1*) (*opt-MatchAny-match-expr m2*)

lemma *opt-MatchAny-match-expr-correct*: *matches* γ (*opt-MatchAny-match-expr m*) = *matches* γ *m*
apply(*case-tac* γ, *rename-tac* β α, *clarify*)
apply(*simp add*: *fun-eq-iff*, *clarify*, *rename-tac a p*)
apply(*rule-tac f*=*opt-MatchAny-match-expr* **in** *matches-iff-apply-f*)
apply(*simp*)
apply(*induction m rule*: *opt-MatchAny-match-expr.induct*)
apply(*simp-all add*: *eval-ternary-simps eval-ternary-idempotence-Not*)
done

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

lemma *MatchNot* (*opt-MatchAny-match-expr* (*MatchAnd MatchAny* (*MatchNot MatchAny*))) = *MatchNot* (*MatchNot MatchAny*) **by** *simp*

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

definition *wf-unknown-match-tac* :: *'p unknown-match-tac* ⇒ *bool* **where**
wf-unknown-match-tac α ≡ (α *Drop* = α *Reject*)

lemma *wf-unknown-match-tacD-False1*: *wf-unknown-match-tac* α ⟹ ¬ *matches* (β, α) *m Reject p* ⟹ *matches* (β, α) *m Drop p* ⟹ *False*
apply(*simp add*: *wf-unknown-match-tac-def*)
apply(*simp add*: *matches-def*)
apply(*case-tac* (*ternary-ternary-eval* (*map-match-tac* β *p m*)))
apply(*simp*)
apply(*simp*)

apply(*simp*)
done

lemma *wf-unknown-match-tacD-False2*: *wf-unknown-match-tac* $\alpha \Longrightarrow$ *matches* (β, α) *m Reject p* $\Longrightarrow \neg$ *matches* (β, α) *m Drop p* $\Longrightarrow$ *False*
apply(*simp add*: *wf-unknown-match-tac-def*)
apply(*simp add*: *matches-def*)
apply(*case-tac* (*ternary-ternary-eval* (*map-match-tac* β *p m*)))
  apply(*simp*)
 apply(*simp*)
apply(*simp*)
done

thm *eval-ternary-simps-simple*

5.2 Removing Unknown Primitives

definition *unknown-match-all* :: *'a unknown-match-tac* $\Rightarrow$ *action* $\Rightarrow$ *bool* **where**
  unknown-match-all α *a* = ($\forall$ *p*. α *a p*)
definition *unknown-not-match-any* :: *'a unknown-match-tac* $\Rightarrow$ *action* $\Rightarrow$ *bool* **where**
  unknown-not-match-any α *a* = ($\forall$ *p*. $\neg$ α *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*. *ternary-ternary-eval* (*map-match-tac* β *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 (β, α) *a* (*MatchNot* (*Match A*)) = (*if*
    ($\forall$ *p*. *ternary-ternary-eval* (*map-match-tac* β *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 MatchNot* (*Match A*)
    else MatchNot (*Match A*)) |
 remove-unknowns-generic (β, α) *a* (*MatchNot* (*MatchNot m*)) = *remove-unknowns-generic* (β, α) *a m* |
  remove-unknowns-generic (β, α) *a* (*MatchAnd m1 m2*) = *MatchAnd*
      (*remove-unknowns-generic* (β, α) *a m1*)
      (*remove-unknowns-generic* (β, α) *a m2*) |

  — $\neg$ (*a* $\wedge$ *b*) = $\neg$ *b* $\vee$ $\neg$ *a* and $\neg$ *Unknown* = *Unknown*
  remove-unknowns-generic (β, α) *a* (*MatchNot* (*MatchAnd m1 m2*)) =
    (*if* (*remove-unknowns-generic* (β, α) *a* (*MatchNot m1*)) = *MatchAny* $\vee$

(*remove-unknowns-generic* (β, α) *a* (*MatchNot m2*)) = *MatchAny*
then MatchAny else
(*if* (*remove-unknowns-generic* (β, α) *a* (*MatchNot m1*)) = *MatchNot MatchAny then*
remove-unknowns-generic (β, α) *a* (*MatchNot m2*) *else*
if (*remove-unknowns-generic* (β, α) *a* (*MatchNot m2*)) = *MatchNot MatchAny then*
remove-unknowns-generic (β, α) *a* (*MatchNot m1*) *else*
MatchNot (*MatchAnd* (*MatchNot* (*remove-unknowns-generic* (β, α) *a* (*MatchNot m1*))) (*MatchNot* (*remove-unknowns-generic* (β, α) *a* (*MatchNot m2*)))))
)

lemma[*code-unfold*]: *remove-unknowns-generic* γ *a* (*MatchNot* (*MatchAnd m1 m2*)) =
(*let m1'* = *remove-unknowns-generic* γ *a* (*MatchNot m1*); *m2'* = *remove-unknowns-generic* γ *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'*)))
)
by(*cases* γ)(*simp*)

lemma *remove-unknowns-generic-simp-3-4-unfolded*: *remove-unknowns-generic* (β, α) *a* (*Match A*) = (*if*
($\forall$ *p*. *ternary-ternary-eval* (*map-match-tac* β *p* (*Match A*)) = *TernaryUnknown*)
then
if ($\forall$ *p*. α *a p*) *then MatchAny else if* ($\forall$ *p*. $\neg$ α *a p*) *then MatchNot MatchAny else Match A*
else (*Match A*))
remove-unknowns-generic (β, α) *a* (*MatchNot* (*Match A*)) = (*if*
($\forall$ *p*. *ternary-ternary-eval* (*map-match-tac* β *p* (*Match A*)) = *TernaryUnknown*)
then
if ($\forall$ *p*. α *a p*) *then MatchAny else if* ($\forall$ *p*. $\neg$ α *a p*) *then MatchNot MatchAny else MatchNot* (*Match A*)
else MatchNot (*Match A*))
by(*auto simp add*: *unknown-match-all-def unknown-not-match-any-def*)

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 $a = Accept \vee a = Drop \Longrightarrow matches\ (\beta,\ \alpha)\ (remove\text{-}unknowns\text{-}generic$
$(\beta,\ \alpha)\ a\ (MatchNot\ (Match\ A)))\ a\ p = matches\ (\beta,\ \alpha)\ (MatchNot\ (Match\ A))\ a$
p
apply(*simp del*: *remove-unknowns-generic.simps add*: *remove-unknowns-generic-simps2*)
apply(*simp add*: *bunch-of-lemmata-about-matches matches-case-ternaryvalue-tuple*)
by *presburger*

lemma *remove-unknowns-generic*: $a = Accept \vee a = Drop \Longrightarrow$
$matches\ \gamma\ (remove\text{-}unknowns\text{-}generic\ \gamma\ a\ m)\ a = matches\ \gamma\ m\ a$
apply(*simp add*: *fun-eq-iff*, *clarify*)
apply(*rename-tac p*)
apply(*induction* γ *a m rule*: *remove-unknowns-generic.induct*)
apply(*simp-all add*: *bunch-of-lemmata-about-matches*)[*2*]
apply(*simp-all add*: *bunch-of-lemmata-about-matches del*: *remove-unknowns-generic.simps add*: *remove-unknowns-generic-simps2*)[*1*]
apply(*simp add*: *matches-case-ternaryvalue-tuple del*: *remove-unknowns-generic.simps add*: *remove-unknowns-generic-simps2*)
apply(*simp-all add*: *bunch-of-lemmata-about-matches matches-DeMorgan*)
apply(*simp-all add*: *matches-case-ternaryvalue-tuple*)
apply *safe*
apply(*simp-all add* : *ternary-to-bool-Some ternary-to-bool-None*)
done

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

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

lemma *packet-independent-unknown-match*: $a = Accept \vee a = Drop \Longrightarrow packet\text{-}independent\text{-}\alpha\ \alpha \Longrightarrow \neg\ unknown\text{-}not\text{-}match\text{-}any\ \alpha\ a \longleftrightarrow unknown\text{-}match\text{-}all\ \alpha\ a$
by(*auto simp add*: *packet-independent-*α*-def unknown-match-all-def unknown-not-match-any-def*)

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* $\Rightarrow$ *bool*

where
  packet-independent-β-unknown $\beta \equiv \forall A.\ (\exists p.\ \beta\ A\ p \neq \mathit{TernaryUnknown}) \longrightarrow (\forall p.\ \beta\ A\ p \neq \mathit{TernaryUnknown})$

lemma *remove-unknowns-generic-specification*: $a = \mathit{Accept} \lor a = \mathit{Drop} \Longrightarrow$ *packet-independent-α* $\alpha \Longrightarrow$ *packet-independent-β-unknown* $\beta \Longrightarrow$
  $\neg$ *has-unknowns* β (*remove-unknowns-generic* (β, α) *a m*)
  proof(*induction* (β, α) *a m rule*: *remove-unknowns-generic.induct*)
 case *3* **thus** *?case* **by**(*simp add*: *packet-independent-unknown-match packet-independent-β-unknown-def*)
  next
 case *4* **thus** *?case* **by**(*simp add*: *packet-independent-unknown-match packet-independent-β-unknown-def*)
  qed(*simp-all*)

end
theory *Semantics-Ternary*
imports *Matching-Ternary ../Misc*
begin

6 Embedded Ternary-Matching Big Step Semantics

lemma *rules-singleton-rev-E*: $[\mathit{Rule}\ m\ a] = rs_1\ @\ rs_2 \Longrightarrow (rs_1 = [\mathit{Rule}\ m\ a] \Longrightarrow rs_2 = [] \Longrightarrow P\ m\ a) \Longrightarrow (rs_1 = [] \Longrightarrow rs_2 = [\mathit{Rule}\ m\ a] \Longrightarrow P\ m\ a) \Longrightarrow P\ m\ a$
by (*cases* rs_1) *auto*

inductive *approximating-bigstep* :: $('a, 'p)$ *match-tac* $\Rightarrow$ $'p$ $\Rightarrow$ $'a$ *rule list* $\Rightarrow$ *state* $\Rightarrow$ *state* $\Rightarrow$ *bool*
  $(\text{-},\text{-}\vdash \langle\text{-}, \text{-}\rangle \Rightarrow_\alpha \text{-}\ \ [60,60,20,98,98]\ 89)$
  for γ **and** p **where**
skip: $\gamma,p\vdash \langle[], t\rangle \Rightarrow_\alpha t$ |
accept: $[\![\mathit{matches}\ \gamma\ m\ \mathit{Accept}\ p]\!] \Longrightarrow \gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Accept}], \mathit{Undecided}\rangle \Rightarrow_\alpha \mathit{Decision}\ \mathit{FinalAllow}$ |
drop: $[\![\mathit{matches}\ \gamma\ m\ \mathit{Drop}\ p]\!] \Longrightarrow \gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Drop}], \mathit{Undecided}\rangle \Rightarrow_\alpha \mathit{Decision}\ \mathit{FinalDeny}$ |
reject: $[\![\mathit{matches}\ \gamma\ m\ \mathit{Reject}\ p]\!] \Longrightarrow \gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Reject}], \mathit{Undecided}\rangle \Rightarrow_\alpha \mathit{Decision}\ \mathit{FinalDeny}$ |
log: $[\![\mathit{matches}\ \gamma\ m\ \mathit{Log}\ p]\!] \Longrightarrow \gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Log}], \mathit{Undecided}\rangle \Rightarrow_\alpha \mathit{Undecided}$ |
empty: $[\![\mathit{matches}\ \gamma\ m\ \mathit{Empty}\ p]\!] \Longrightarrow \gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Empty}], \mathit{Undecided}\rangle \Rightarrow_\alpha \mathit{Undecided}$ |
nomatch: $[\![\neg\ \mathit{matches}\ \gamma\ m\ a\ p]\!] \Longrightarrow \gamma,p\vdash \langle[\mathit{Rule}\ m\ a], \mathit{Undecided}\rangle \Rightarrow_\alpha \mathit{Undecided}$
|
decision: $\gamma,p\vdash \langle rs, \mathit{Decision}\ X\rangle \Rightarrow_\alpha \mathit{Decision}\ X$ |
seq: $[\![\gamma,p\vdash \langle rs_1, \mathit{Undecided}\rangle \Rightarrow_\alpha t;\ \gamma,p\vdash \langle rs_2, t\rangle \Rightarrow_\alpha t']\!] \Longrightarrow \gamma,p\vdash \langle rs_1 @ rs_2, \mathit{Undecided}\rangle \Rightarrow_\alpha t'$

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*] : γ,*p*⊢ ⟨*rs*,*s*⟩ $\Rightarrow_\alpha$ *t* $\Longrightarrow$
($\bigwedge$*t*. *P* [] *t t*) $\Longrightarrow$
($\bigwedge$*m a*. *matches* γ *m a p* $\Longrightarrow$ *a* = *Accept* $\Longrightarrow$ *P* [*Rule m a*] *Undecided* (*Decision FinalAllow*)) $\Longrightarrow$
($\bigwedge$*m a*. *matches* γ *m a p* $\Longrightarrow$ *a* = *Drop* $\vee$ *a* = *Reject* $\Longrightarrow$ *P* [*Rule m a*] *Undecided* (*Decision FinalDeny*)) $\Longrightarrow$
($\bigwedge$*m a*. *matches* γ *m a p* $\Longrightarrow$ *a* = *Log* $\vee$ *a* = *Empty* $\Longrightarrow$ *P* [*Rule m a*] *Undecided Undecided*) $\Longrightarrow$
($\bigwedge$*m a*. $\neg$ *matches* γ *m a p* $\Longrightarrow$ *P* [*Rule m a*] *Undecided Undecided*) $\Longrightarrow$
($\bigwedge$*rs X*. *P rs* (*Decision X*) (*Decision X*)) $\Longrightarrow$
($\bigwedge$*rs* rs_1 rs_2 *t t'*. *rs* = rs_1 @ rs_2 $\Longrightarrow$ γ,*p*⊢ ⟨rs_1,*Undecided*⟩ $\Rightarrow_\alpha$ *t* $\Longrightarrow$ *P* rs_1 *Undecided t* $\Longrightarrow$ γ,*p*⊢ ⟨rs_2,*t*⟩ $\Rightarrow_\alpha$ *t'* $\Longrightarrow$ *P* rs_2 *t t'* $\Longrightarrow$ *P rs Undecided t'*)
$\Longrightarrow$ *P rs s t*
by (*induction rule*: *approximating-bigstep.induct*) (*simp-all*)

lemma *skipD*: γ,*p*⊢ ⟨[], *s*⟩ $\Rightarrow_\alpha$ *t* $\Longrightarrow$ *s* = *t*
by (*induction* []::'*a rule list s t rule*: *approximating-bigstep-induct*) (*simp-all*)

lemma *decisionD*: γ,*p*⊢ ⟨*rs*, *Decision X*⟩ $\Rightarrow_\alpha$ *t* $\Longrightarrow$ *t* = *Decision X*
by (*induction rs Decision X t rule*: *approximating-bigstep-induct*) (*simp-all*)

lemma *acceptD*: γ,*p*⊢ ⟨[*Rule m Accept*], *Undecided*⟩ $\Rightarrow_\alpha$ *t* $\Longrightarrow$ *matches* γ *m Accept p* $\Longrightarrow$ *t* = *Decision FinalAllow*
apply (*induction* [*Rule m Accept*] *Undecided t rule*: *approximating-bigstep-induct*)
  apply (*simp-all*)
by (*metis list-app-singletonE skipD*)

lemma *dropD*: γ,*p*⊢ ⟨[*Rule m Drop*], *Undecided*⟩ $\Rightarrow_\alpha$ *t* $\Longrightarrow$ *matches* γ *m Drop p* $\Longrightarrow$ *t* = *Decision FinalDeny*
apply (*induction* [*Rule m Drop*] *Undecided t rule*: *approximating-bigstep-induct*)
by(*auto dest*: *skipD elim*!: *rules-singleton-rev-E*)

lemma *rejectD*: γ,*p*⊢ ⟨[*Rule m Reject*], *Undecided*⟩ $\Rightarrow_\alpha$ *t* $\Longrightarrow$ *matches* γ *m Reject p* $\Longrightarrow$ *t* = *Decision FinalDeny*
apply (*induction* [*Rule m Reject*] *Undecided t rule*: *approximating-bigstep-induct*)
by(*auto dest*: *skipD elim*!: *rules-singleton-rev-E*)

lemma *logD*: γ,*p*⊢ ⟨[*Rule m Log*], *Undecided*⟩ $\Rightarrow_\alpha$ *t* $\Longrightarrow$ *t* = *Undecided*
apply (*induction* [*Rule m Log*] *Undecided t rule*: *approximating-bigstep-induct*)
by(*auto dest*: *skipD elim*!: *rules-singleton-rev-E*)

lemma *emptyD*: γ,*p*⊢ ⟨[*Rule m Empty*], *Undecided*⟩ $\Rightarrow_\alpha$ *t* $\Longrightarrow$ *t* = *Undecided*

apply (*induction* [*Rule m Empty*] *Undecided t rule*: *approximating-bigstep-induct*)
by(*auto dest*: *skipD elim*!: *rules-singleton-rev-E*)

lemma *nomatchD*: γ,*p*⊢ ⟨[*Rule m a*], *Undecided*⟩ $\Rightarrow_\alpha$ *t* $\Longrightarrow$ ¬ *matches* γ *m a p* $\Longrightarrow$ *t* = *Undecided*
apply (*induction* [*Rule m a*] *Undecided t rule*: *approximating-bigstep-induct*)
by(*auto dest*: *skipD elim*!: *rules-singleton-rev-E*)

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

lemma *approximating-bigstep-to-undecided*: γ,*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *Undecided* $\Longrightarrow$ *s* = *Undecided*
  by (*metis decisionD state.exhaust*)

lemma *approximating-bigstep-to-decision1*: γ,*p*⊢ ⟨*rs*, *Decision Y*⟩ $\Rightarrow_\alpha$ *Decision X* $\Longrightarrow$ *Y* = *X*
  by (*metis decisionD state.inject*)
thm *decisionD*

lemma *nomatch-fst*: ¬ *matches* γ *m a p* $\Longrightarrow$ γ,*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *t* $\Longrightarrow$ γ,*p*⊢ ⟨*Rule m a* # *rs*, *s*⟩ $\Rightarrow_\alpha$ *t*
  apply(*cases s*)
   apply(*clarify*)
   apply(*drule nomatch*)
   apply(*drule*(*1*) *seq*)
   apply (*simp*)
  apply(*clarify*)
  apply(*drule decisionD*)
  apply(*clarify*)
 apply(*simp-all add*: *decision*)
done

lemma *seq'*:
  assumes *rs* = rs_1 @ rs_2 γ,*p*⊢ ⟨rs_1,*s*⟩ $\Rightarrow_\alpha$ *t* γ,*p*⊢ ⟨rs_2,*t*⟩ $\Rightarrow_\alpha$ *t'*
  shows γ,*p*⊢ ⟨*rs*,*s*⟩ $\Rightarrow_\alpha$ *t'*
using *assms* **by** (*cases s*) (*auto intro*: *seq decision dest*: *decisionD*)

lemma *seq-split*:
  assumes γ,*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *t* *rs* = rs_1@rs_2
  obtains *t'* **where** γ,*p*⊢ ⟨rs_1,*s*⟩ $\Rightarrow_\alpha$ *t'* γ,*p*⊢ ⟨rs_2,*t'*⟩ $\Rightarrow_\alpha$ *t*
  using *assms*
 proof (*induction rs s t arbitrary*: rs_1 rs_2 *thesis rule*: *approximating-bigstep-induct*)
   case *Allow* **thus** *?case* **by** (*auto dest*: *skipD elim*!: *rules-singleton-rev-E intro*: *approximating-bigstep.intros*)
  next
   case *Deny* **thus** *?case* **by** (*auto dest*: *skipD elim*!: *rules-singleton-rev-E intro*: *approximating-bigstep.intros*)
  next

case *Log* **thus** *?case* **by** (*auto dest*: *skipD elim*!: *rules-singleton-rev-E intro*: *approximating-bigstep.intros*)
next
case *Nomatch* **thus** *?case* **by** (*auto dest*: *skipD elim*!: *rules-singleton-rev-E intro*: *approximating-bigstep.intros*)
next
case (*Seq rs rsa rsb t t'*)
hence *rs*: $rsa @ rsb = rs_1 @ rs_2$ **by** *simp*
note *List.append-eq-append-conv-if* [*simp*]
from *rs* **show** *?case*
proof (*cases rule*: *list-app-eq-cases*)
case *longer*
with *Seq* **have** *t1*: $\gamma,p\vdash \langle take\ (length\ rsa)\ rs_1,\ Undecided\rangle \Rightarrow_\alpha t$
by *simp*
from *Seq longer* **obtain** *t2*
where *t2a*: $\gamma,p\vdash \langle drop\ (length\ rsa)\ rs_1,t\rangle \Rightarrow_\alpha t2$
and *rs2-t2*: $\gamma,p\vdash \langle rs_2,t2\rangle \Rightarrow_\alpha t'$
by *blast*
with *t1 rs2-t2* **have** $\gamma,p\vdash \langle take\ (length\ rsa)\ rs_1 @ drop\ (length\ rsa)\ rs_1,Undecided\rangle \Rightarrow_\alpha t2$
by (*blast intro*: *approximating-bigstep.seq*)
with *Seq rs2-t2* **show** *?thesis*
by *simp*
next
case *shorter*
with *rs* **have** *rsa'*: $rsa = rs_1 @ take\ (length\ rsa - length\ rs_1)\ rs_2$
by (*metis append-eq-conv-conj length-drop*)
from *shorter rs* **have** *rsb'*: $rsb = drop\ (length\ rsa - length\ rs_1)\ rs_2$
by (*metis append-eq-conv-conj length-drop*)
from *Seq rsa'* **obtain** *t1*
where *t1a*: $\gamma,p\vdash \langle rs_1,Undecided\rangle \Rightarrow_\alpha t1$
and *t1b*: $\gamma,p\vdash \langle take\ (length\ rsa - length\ rs_1)\ rs_2,t1\rangle \Rightarrow_\alpha t$
by *blast*
from *rsb' Seq.hyps* **have** *t2*: $\gamma,p\vdash \langle drop\ (length\ rsa - length\ rs_1)\ rs_2,t\rangle \Rightarrow_\alpha t'$
by *blast*
with *seq' t1b* **have** $\gamma,p\vdash \langle rs_2,t1\rangle \Rightarrow_\alpha t'$ **by** (*metis append-take-drop-id*)
with *Seq t1a* **show** *?thesis*
by *fast*
qed
qed (*auto intro*: *approximating-bigstep.intros*)

lemma *seqE-fst*:
assumes $\gamma,p\vdash \langle r\#rs,\ s\rangle \Rightarrow_\alpha t$
obtains t' **where** $\gamma,p\vdash \langle [r],s\rangle \Rightarrow_\alpha t'$ $\gamma,p\vdash \langle rs,t'\rangle \Rightarrow_\alpha t$
using *assms seq-split* **by** (*metis append-Cons append-Nil*)

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(*cases s*)
  case *Undecided* **with** *assms seq* **show** $\gamma,p\vdash \langle r \# rs, s\rangle \Rightarrow_\alpha t'$ **by** *fastforce*
  next
  case *Decision* **with** *assms* **show** $\gamma,p\vdash \langle r \# rs, s\rangle \Rightarrow_\alpha t'$
  by(*auto simp*: *decision dest*!: *decisionD*)
qed

fun *approximating-bigstep-fun* :: ('*a*, '*p*) *match-tac* ⇒ '*p* ⇒ '*a rule list* ⇒ *state* ⇒ *state* **where**
  approximating-bigstep-fun γ *p* [] *s* = *s* |
  approximating-bigstep-fun γ *p rs* (*Decision X*) = (*Decision X*) |
  approximating-bigstep-fun γ *p* ((*Rule m a*)#*rs*) *Undecided* = (*if*
    ¬ *matches* γ *m a p*
   then
    approximating-bigstep-fun γ *p rs Undecided*
   else
    case a of Accept ⇒ *Decision FinalAllow*
        | *Drop* ⇒ *Decision FinalDeny*
        | *Reject* ⇒ *Decision FinalDeny*
        | *Log* ⇒ *approximating-bigstep-fun* γ *p rs Undecided*
        | *Empty* ⇒ *approximating-bigstep-fun* γ *p rs Undecided*
        (**unhalndled cases**)
)

lemma *approximating-bigstep-fun-induct*[*case-names Empty Decision Nomatch Match*]
:
$(\bigwedge \gamma\ p\ s.\ P\ \gamma\ p\ []\ s) \Longrightarrow$
$(\bigwedge \gamma\ p\ r\ rs\ X.\ P\ \gamma\ p\ (r \# rs)\ (Decision\ X)) \Longrightarrow$
$(\bigwedge \gamma\ p\ m\ a\ rs.$
  $\neg\ matches\ \gamma\ m\ a\ p \Longrightarrow P\ \gamma\ p\ rs\ Undecided \Longrightarrow P\ \gamma\ p\ (Rule\ m\ a \#\ rs)$
$Undecided) \Longrightarrow$
$(\bigwedge \gamma\ p\ m\ a\ rs.$
  $matches\ \gamma\ m\ a\ p \Longrightarrow (a = Log \Longrightarrow P\ \gamma\ p\ rs\ Undecided) \Longrightarrow (a = Empty \Longrightarrow$
$P\ \gamma\ p\ rs\ Undecided) \Longrightarrow P\ \gamma\ p\ (Rule\ m\ a \#\ rs)\ Undecided) \Longrightarrow$
$P\ \gamma\ p\ rs\ s$
apply (*rule approximating-bigstep-fun.induct*[*of P* γ *p rs s*])
  apply (*simp-all*)
by *metis*

lemma *Decision-approximating-bigstep-fun*: *approximating-bigstep-fun* γ *p rs* (*Decision X*) = *Decision X*
  by(*induction rs*) (*simp-all*)

6.1 wf ruleset

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

1. either the rules do not match
2. 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-ruleset γ *p rs* $\equiv \forall r \in$ *set rs*.
($\neg$ *matches* γ (*get-match r*) (*get-action r*) *p*) $\vee$
($\neg(\exists$ *chain*. *get-action r* = *Call chain*) $\wedge$ *get-action r* $\neq$ *Return* $\wedge$ *get-action r* $\neq$ *Unknown*)

lemma *wf-ruleset-append*: *wf-ruleset* γ *p* (*rs1@rs2*) $\longleftrightarrow$ *wf-ruleset* γ *p rs1* $\wedge$ *wf-ruleset* γ *p rs2*
by(*auto simp add*: *wf-ruleset-def*)
lemma *wf-rulesetD*: **assumes** *wf-ruleset* γ *p* (*r* # *rs*) **shows** *wf-ruleset* γ *p* [*r*] **and** *wf-ruleset* γ *p rs*
using *assms* **by**(*auto simp add*: *wf-ruleset-def*)
lemma *wf-ruleset-fst*: *wf-ruleset* γ *p* (*Rule m a* # *rs*) $\longleftrightarrow$ *wf-ruleset* γ *p* [*Rule m a*] $\wedge$ *wf-ruleset* γ *p rs*
using *assms* **by**(*auto simp add*: *wf-ruleset-def*)
lemma *wf-ruleset-stripfst*: *wf-ruleset* γ *p* (*r* # *rs*) $\Longrightarrow$ *wf-ruleset* γ *p* (*rs*)
by(*simp add*: *wf-ruleset-def*)
lemma *wf-ruleset-rest*: *wf-ruleset* γ *p* (*Rule m a* # *rs*) $\Longrightarrow$ *wf-ruleset* γ *p* [*Rule m a*]
by(*simp add*: *wf-ruleset-def*)

lemma *approximating-bigstep-fun-induct-wf*[*case-names Empty Decision Nomatch MatchAccept MatchDrop MatchReject MatchLog MatchEmpty*, *consumes 1*]:
wf-ruleset γ *p rs* $\Longrightarrow$
($\bigwedge\gamma$ *p s*. *P* γ *p* [] *s*) $\Longrightarrow$
($\bigwedge\gamma$ *p r rs X*. *P* γ *p* (*r* # *rs*) (*Decision X*)) $\Longrightarrow$
($\bigwedge\gamma$ *p m a rs*.
$\neg$ *matches* γ *m a p* $\Longrightarrow$ *P* γ *p rs Undecided* $\Longrightarrow$ *P* γ *p* (*Rule m a* # *rs*) *Undecided*) $\Longrightarrow$
($\bigwedge\gamma$ *p m a rs*.
matches γ *m a p* $\Longrightarrow$ *a* = *Accept* $\Longrightarrow$ *P* γ *p* (*Rule m a* # *rs*) *Undecided*) $\Longrightarrow$
($\bigwedge\gamma$ *p m a rs*.
matches γ *m a p* $\Longrightarrow$ *a* = *Drop* $\Longrightarrow$ *P* γ *p* (*Rule m a* # *rs*) *Undecided*) $\Longrightarrow$
($\bigwedge\gamma$ *p m a rs*.
matches γ *m a p* $\Longrightarrow$ *a* = *Reject* $\Longrightarrow$ *P* γ *p* (*Rule m a* # *rs*) *Undecided*) $\Longrightarrow$
($\bigwedge\gamma$ *p m a rs*.
matches γ *m a p* $\Longrightarrow$ *a* = *Log* $\Longrightarrow$ *P* γ *p rs Undecided* $\Longrightarrow$ *P* γ *p* (*Rule m a* # *rs*) *Undecided*) $\Longrightarrow$
($\bigwedge\gamma$ *p m a rs*.
matches γ *m a p* $\Longrightarrow$ *a* = *Empty* $\Longrightarrow$ *P* γ *p rs Undecided* $\Longrightarrow$ *P* γ *p* (*Rule m a* # *rs*) *Undecided*) $\Longrightarrow$

$P\ \gamma\ p\ rs\ s$
proof(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)
case *Empty* **thus** *?case* **by** *blast*
next
case *Decision* **thus** *?case* **by** *blast*
next
case *Nomatch* **thus** *?case* **by**(*simp add*: *wf-ruleset-def*)
next
case (*Match* γ *p m a*) **thus** *?case*
apply −
apply(*frule wf-rulesetD*(*1*), *drule wf-rulesetD*(*2*))
apply(*simp*)
apply(*cases a*)
apply(*simp-all*)
apply(*auto simp add*: *wf-ruleset-def*)
done
qed

6.1.1 Append, Prepend, Postpend, Composition

lemma *approximating-bigstep-fun-seq-wf*: $[\![$ *wf-ruleset* γ *p* $rs_1]\!] \Longrightarrow$
approximating-bigstep-fun γ *p* (rs_1 @ rs_2) *s* = *approximating-bigstep-fun* γ *p* rs_2 (*approximating-bigstep-fun* γ *p* rs_1 *s*)
proof(*induction* γ *p* rs_1 *s rule*: *approximating-bigstep-fun-induct*)
qed(*simp-all add*: *wf-ruleset-def Decision-approximating-bigstep-fun split*: *action.split*)

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

lemma *approximating-bigstep-fun-seq-Undecided-wf*: $[\![$ *wf-ruleset* γ *p* (*rs1*@*rs2*)$]\!]$ $\Longrightarrow$
approximating-bigstep-fun γ *p* (*rs1*@*rs2*) *Undecided* = *Undecided* $\longleftrightarrow$
approximating-bigstep-fun γ *p rs1 Undecided* = *Undecided* $\wedge$ *approximating-bigstep-fun* γ *p rs2 Undecided* = *Undecided*
proof(*induction* γ *p rs1 Undecided rule*: *approximating-bigstep-fun-induct*)
qed(*simp-all add*: *wf-ruleset-def split*: *action.split*)

lemma *approximating-bigstep-fun-seq-Undecided-t-wf*: $[\![$ *wf-ruleset* γ *p* (*rs1*@*rs2*)$]\!]$ $\Longrightarrow$
approximating-bigstep-fun γ *p* (*rs1*@*rs2*) *Undecided* = *t* $\longleftrightarrow$
approximating-bigstep-fun γ *p rs1 Undecided* = *Undecided* $\wedge$ *approximating-bigstep-fun* γ *p rs2 Undecided* = *t* $\vee$
approximating-bigstep-fun γ *p rs1 Undecided* = *t* $\wedge$ *t* $\neq$ *Undecided*
proof(*induction* γ *p rs1 Undecided rule*: *approximating-bigstep-fun-induct*)
case *Empty* **thus** *?case* **by**(*cases t*) *simp-all*
next
case *Nomatch* **thus** *?case* **by**(*simp add*: *wf-ruleset-def*)
next

case *Match* **thus** *?case* **by**(*auto simp add*: *wf-ruleset-def split*: *action.split*)
qed

lemma *approximating-bigstep-fun-wf-postpend*: *wf-ruleset* γ *p rsA* $\Longrightarrow$ *wf-ruleset* γ *p rsB* $\Longrightarrow$
approximating-bigstep-fun γ *p rsA s* = *approximating-bigstep-fun* γ *p rsB s* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*rsA@rsC*) *s* = *approximating-bigstep-fun* γ *p* (*rsB@rsC*) *s*
apply(*induction* γ *p rsA s rule*: *approximating-bigstep-fun-induct-wf*)
apply(*simp-all add*: *approximating-bigstep-fun-seq-wf*)
apply (*metis Decision-approximating-bigstep-fun*)+
done

lemma *approximating-bigstep-fun-singleton-prepend*:
assumes *approximating-bigstep-fun* γ *p rsB s* = *approximating-bigstep-fun* γ *p rsC s*
shows *approximating-bigstep-fun* γ *p* (*r#rsB*) *s* = *approximating-bigstep-fun* γ *p* (*r#rsC*) *s*
proof(*cases s*)
case *Decision* **thus** *?thesis* **by**(*simp add*: *Decision-approximating-bigstep-fun*)
next
case *Undecided*
with *assms* **show** *?thesis* **by**(*cases r*)(*simp split*: *action.split*)
qed

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

lemma *approximating-bigstep-wf*: $\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow_\alpha Undecided \Longrightarrow$ *wf-ruleset* γ *p rs*
unfolding *wf-ruleset-def*
proof(*induction rs Undecided Undecided rule*: *approximating-bigstep-induct*)
case *Skip* **thus** *?case* **by** *simp*
next
case *Log* **thus** *?case* **by** *auto*
next
case *Nomatch* **thus** *?case* **by** *simp*
next
case (*Seq rs rs1 rs2 t*)
from *Seq approximating-bigstep-to-undecided* **have** *t* = *Undecided* **by** *fast*
from *this Seq* **show** *?case* **by** *auto*
qed

only valid actions appear in this ruleset

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

lemma[*code-unfold*]: *good-ruleset rs* = ($\forall$ *r*$\in$*set rs*. (*case get-action r of Call chain* $\Rightarrow$ *False* | *Return* $\Rightarrow$ *False* | *Unknown* $\Rightarrow$ *False* | - $\Rightarrow$ *True*))
unfolding *good-ruleset-def*
apply(*rule Set.ball-cong*)
apply(*simp-all*)
apply(*rename-tac r*)
by(*case-tac get-action r*)(*simp-all*)

lemma *good-ruleset-alt*: *good-ruleset rs* = ($\forall$ *r*$\in$*set rs*. *get-action r* = *Accept* $\vee$ *get-action r* = *Drop* $\vee$
get-action r = *Reject* $\vee$ *get-action r* = *Log* $\vee$ *get-action r* = *Empty*)
unfolding *good-ruleset-def*
apply(*rule Set.ball-cong*)
apply(*simp-all*)
apply(*rename-tac r*)
by(*case-tac get-action r*)(*simp-all*)

lemma *good-ruleset-append*: *good-ruleset* (rs_1 @ rs_2) $\longleftrightarrow$ *good-ruleset* rs_1 $\wedge$ *good-ruleset* rs_2
by(*simp add*: *good-ruleset-alt*, *blast*)

lemma *good-ruleset-fst*: *good-ruleset* (*r*#*rs*) $\Longrightarrow$ *good-ruleset* [*r*]
by(*simp add*: *good-ruleset-def*)
lemma *good-ruleset-tail*: *good-ruleset* (*r*#*rs*) $\Longrightarrow$ *good-ruleset rs*
by(*simp add*: *good-ruleset-def*)

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

lemma *good-imp-wf-ruleset*: *good-ruleset rs* $\Longrightarrow$ *wf-ruleset* γ *p rs* **by** (*metis good-ruleset-def wf-ruleset-def*)

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* = *Reject**) $\vee$ *get-action r* = *Drop*
lemma *simple-imp-good-ruleset*: *simple-ruleset rs* $\Longrightarrow$ *good-ruleset rs*
by(*simp add*: *simple-ruleset-def good-ruleset-def*, *fastforce*)

lemma *simple-ruleset-tail*: *simple-ruleset* (*r*#*rs*) $\Longrightarrow$ *simple-ruleset rs* **by** (*simp add*: *simple-ruleset-def*)

lemma *simple-ruleset-append*: *simple-ruleset* (rs_1 @ rs_2) $\longleftrightarrow$ *simple-ruleset* rs_1 $\wedge$ *simple-ruleset* rs_2
by(*simp add*: *simple-ruleset-def*, *blast*)

lemma *approximating-bigstep-fun-seq-semantics*: $[\![$ γ,*p*$\vdash$ $\langle rs_1, s\rangle \Rightarrow_\alpha t$ $]\!]$ $\Longrightarrow$
approximating-bigstep-fun γ *p* (rs_1 @ rs_2) *s* = *approximating-bigstep-fun* γ *p*

rs_2 t
  proof(*induction* rs_1 *s t arbitrary*: rs_2 *rule*: *approximating-bigstep.induct*)
  qed(*simp-all add*: *Decision-approximating-bigstep-fun*)

lemma *approximating-semantics-imp-fun*: γ,*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *t* ⟹ *approximating-bigstep-fun* γ *p rs s* = *t*
  proof(*induction rs s t rule*: *approximating-bigstep-induct*)
  qed(*auto simp add*: *approximating-bigstep-fun-seq-semantics Decision-approximating-bigstep-fun*)

lemma *approximating-fun-imp-semantics*: **assumes** *wf-ruleset* γ *p rs*
      shows *approximating-bigstep-fun* γ *p rs s* = *t* ⟹ γ,*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *t*
  using *assms* **proof**(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct-wf*)
    case (*Empty* γ *p s*)
      thus γ,*p*⊢ ⟨[], *s*⟩ $\Rightarrow_\alpha$ *t* **using** *skip* **by**(*simp*)
    next
    case (*Decision* γ *p r rs X*)
      hence *t* = *Decision X* **by** *simp*
      thus γ,*p*⊢ ⟨*r* # *rs*, *Decision X*⟩ $\Rightarrow_\alpha$ *t* **using** *decision* **by** *fast*
    next
    case (*Nomatch* γ *p m a rs*)
      thus γ,*p*⊢ ⟨*Rule m a* # *rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *t*
        apply(*rule-tac t*=*Undecided* **in** *seq-fst*)
         apply(*simp add*: *nomatch*)
        apply(*simp add*: *Nomatch.IH*)
        done
    next
    case (*MatchAccept* γ *p m a rs*)
      hence *t* = *Decision FinalAllow* **by** *simp*
      thus *?case* **by** (*metis MatchAccept.hyps accept decision seq-fst*)
    next
    case (*MatchDrop* γ *p m a rs*)
      hence *t* = *Decision FinalDeny* **by** *simp*
      thus *?case* **by** (*metis MatchDrop.hyps drop decision seq-fst*)
    next
    case (*MatchReject* γ *p m a rs*)
      hence *t* = *Decision FinalDeny* **by** *simp*
      thus *?case* **by** (*metis MatchReject.hyps reject decision seq-fst*)
    next
    case (*MatchLog* γ *p m a rs*)
      thus *?case*
        apply(*simp*)
        apply(*rule-tac t*=*Undecided* **in** *seq-fst*)
         apply(*simp add*: *log*)
        apply(*simp add*: *MatchLog.IH*)
        done
    next
    case (*MatchEmpty* γ *p m a rs*)
      thus *?case*
        apply(*simp*)

apply(*rule-tac t=Undecided* **in** *seq-fst*)
apply(*simp add*: *empty*)
apply(*simp add*: *MatchEmpty.IH*)
done
qed

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

theorem *approximating-semantics-iff-fun*: *wf-ruleset* γ *p rs* $\Longrightarrow$
γ,*p*$\vdash$ $\langle rs,\ s\rangle \Rightarrow_\alpha t \longleftrightarrow$ *approximating-bigstep-fun* γ *p rs s* = *t*
by (*metis approximating-fun-imp-semantics approximating-semantics-imp-fun*)

corollary *approximating-semantics-iff-fun-good-ruleset*: *good-ruleset rs* $\Longrightarrow$
γ,*p*$\vdash$ $\langle rs,\ s\rangle \Rightarrow_\alpha t \longleftrightarrow$ *approximating-bigstep-fun* γ *p rs s* = *t*
by (*metis approximating-semantics-iff-fun good-imp-wf-ruleset*)

lemma *approximating-bigstep-deterministic*: $[\![$ γ,*p*$\vdash$ $\langle rs,\ s\rangle \Rightarrow_\alpha t$; γ,*p*$\vdash$ $\langle rs,\ s\rangle \Rightarrow_\alpha$ t' $]\!] \Longrightarrow t = t'$
proof(*induction arbitrary*: t' *rule*: *approximating-bigstep-induct*)
case *Seq* **thus** *?case*
by (*metis* (*hide-lams*, *mono-tags*) *append-Nil2 approximating-bigstep-fun.simps*(*1*) *approximating-bigstep-fun-seq-semantics*)
qed(*auto dest*: *approximating-bigstepD*)

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-fun-semantics*:
approximating-bigstep-fun γ *p* (*rm-LogEmpty rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)
case *Empty* **thus** *?case* **by**(*simp*)
next
case *Decision* **thus** *?case* **by**(*simp add*: *Decision-approximating-bigstep-fun*)
next
case (*Nomatch* γ *p m a rs*) **thus** *?case* **by**(*cases a*,*simp-all*)
next
case (*Match* γ *p m a rs*) **thus** *?case* **by**(*cases a*,*simp-all*)
qed

lemma *rm-LogEmpty-seq*: *rm-LogEmpty* (*rs1*@*rs2*) = *rm-LogEmpty rs1* @ *rm-LogEmpty rs2*
proof(*induction rs1*)
case *Nil* **thus** *?case* **by** *simp*

next
case (*Cons r rs*) **thus** *?case*
  apply(*cases r*, *rename-tac m a*)
  apply(*simp*)
  apply(*case-tac a*)
    apply(*simp-all*)
  done
qed

lemma γ,*p*⊢ ⟨*rm-LogEmpty rs*, *s*⟩ $\Rightarrow_\alpha$ *t* $\longleftrightarrow$ γ,*p*⊢ ⟨*rs*, *s*⟩ $\Rightarrow_\alpha$ *t*
apply(*rule iffI*)
 apply(*induction rs arbitrary*: *s t*)
  apply(*simp-all*)
 apply(*rename-tac r rs s t*)
 apply(*case-tac r*)
 apply(*simp*)
 apply(*rename-tac m a*)
 apply(*case-tac a*)
  apply(*simp-all*)
  apply(*auto intro*: *approximating-bigstep.intros*)
  apply(*erule seqE-fst*, *simp add*: *seq-fst*)
  apply(*erule seqE-fst*, *simp add*: *seq-fst*)
  apply (*metis decision log nomatch-fst seq-fst state.exhaust*)
  apply(*erule seqE-fst*, *simp add*: *seq-fst*)
  apply(*erule seqE-fst*, *simp add*: *seq-fst*)
  apply(*erule seqE-fst*, *simp add*: *seq-fst*)
  apply (*metis decision empty nomatch-fst seq-fst state.exhaust*)
 apply(*erule seqE-fst*, *simp add*: *seq-fst*)
apply(*induction rs s t rule*: *approximating-bigstep-induct*)
  apply(*auto intro*: *approximating-bigstep.intros*)
 apply(*rename-tac m a*)
 apply(*case-tac a*)
  apply(*auto intro*: *approximating-bigstep.intros*)
apply(*rename-tac* rs_1 rs_2 *t t'*)
apply(*drule-tac* rs_1=*rm-LogEmpty* rs_1 **and** rs_2=*rm-LogEmpty* rs_2 **in** *seq*)
 apply(*simp-all*)
using *rm-LogEmpty-seq* **apply** *metis*
done

lemma *rm-LogEmpty-simple-but-Reject*:
 good-ruleset rs $\Longrightarrow$ $\forall$ *r* $\in$ *set* (*rm-LogEmpty rs*). *get-action r* = *Accept* $\vee$ *get-action r* = *Reject* $\vee$ *get-action r* = *Drop*
 proof(*induction rs*)
 case *Nil* **thus** *?case* **by**(*simp add*: *good-ruleset-def*)
 next
 case (*Cons r rs*) **thus** *?case*

apply(*clarify*)
apply(*cases r*, *rename-tac m a*, *simp*)
by(*case-tac a*) (*auto simp add*: *good-ruleset-def*)
qed

Rewrite *Reject* actions to *Drop* actions

fun *rw-Reject* :: *'a rule list* ⇒ *'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*

lemma *rw-Reject-fun-semantics*:
wf-unknown-match-tac $\alpha \Longrightarrow$
(*approximating-bigstep-fun* (β, α) *p* (*rw-Reject rs*) *s* = *approximating-bigstep-fun* (β, α) *p rs s*)
proof(*induction rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
thus *?case*
apply(*case-tac r*, *rename-tac m a*, *simp*)
apply(*case-tac a*)
apply(*case-tac* [!] *s*)
apply(*auto dest*: *wf-unknown-match-tacD-False1 wf-unknown-match-tacD-False2*)
done
qed

lemma *rmLogEmpty-rwReject-good-to-simple*: *good-ruleset rs* $\Longrightarrow$ *simple-ruleset* (*rw-Reject* (*rm-LogEmpty rs*))
apply(*drule rm-LogEmpty-simple-but-Reject*)
apply(*simp add*: *simple-ruleset-def*)
apply(*induction rs*)
apply(*simp-all*)
apply(*rename-tac r rs*)
apply(*case-tac r*)
apply(*rename-tac m a*)
apply(*case-tac a*)
apply(*simp-all*)
done

definition *optimize-matches* :: (*'a match-expr* ⇒ *'a match-expr*) ⇒ *'a rule list* ⇒ *'a rule list* **where**
optimize-matches f rs = *map* (λr. *Rule* (*f* (*get-match r*)) (*get-action r*)) *rs*

lemma *optimize-matches*: $\forall$ *m*. *matches* γ *m* = *matches* γ (*f m*) $\Longrightarrow$ *approximating-bigstep-fun* γ *p* (*optimize-matches f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)
case (*Match* γ *p m a rs*) **thus** *?case* **by**(*case-tac a*)(*simp-all add*: *optimize-matches-def*)

qed(*simp-all add*: *optimize-matches-def*)

lemma *optimize-matches-simple-ruleset*: *simple-ruleset rs* $\Longrightarrow$ *simple-ruleset* (*optimize-matches f rs*)
by(*simp add*: *optimize-matches-def simple-ruleset-def*)

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*
using *optimize-matches opt-MatchAny-match-expr-correct* **by** *metis*

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* (λr. *Rule* (*f* (*get-action r*) (*get-match r*)) (*get-action r*)) *rs*

lemma *optimize-matches-a-simple-ruleset*: *simple-ruleset rs* $\Longrightarrow$ *simple-ruleset* (*optimize-matches-a f rs*)
by(*simp add*: *optimize-matches-a-def simple-ruleset-def*)

lemma *optimize-matches-a*: $\forall$ *a m*. *matches* γ *m a* = *matches* γ (*f a m*) *a* $\Longrightarrow$ *approximating-bigstep-fun* γ *p* (*optimize-matches-a f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)
case (*Match* γ *p m a rs*) **thus** *?case* **by**(*case-tac a*)(*simp-all add*: *optimize-matches-a-def*)
qed(*simp-all add*: *optimize-matches-a-def*)

lemma *optimize-matches-a-simplers*:
assumes *simple-ruleset rs* **and** $\forall$ *a m*. *a* = *Accept* $\vee$ *a* = *Drop* $\longrightarrow$ *matches* γ (*f a m*) *a* = *matches* γ *m a*
shows *approximating-bigstep-fun* γ *p* (*optimize-matches-a f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof –
from *assms*(*1*) **have** *wf-ruleset* γ *p rs* **by**(*simp add*: *simple-imp-good-ruleset good-imp-wf-ruleset*)
from ‹*wf-ruleset* γ *p rs*› *assms* **show** *approximating-bigstep-fun* γ *p* (*optimize-matches-a f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct-wf*)
case *Nomatch* **thus** *?case*
apply(*simp add*: *optimize-matches-a-def simple-ruleset-def*)
apply(*safe*)
apply(*simp-all*)
done
next
case *MatchReject* **thus** *?case* **by**(*simp add*: *optimize-matches-a-def simple-ruleset-def*)
qed(*simp-all add*: *optimize-matches-a-def simple-ruleset-tail*)
qed

end

theory *Datatype-Selectors*
imports *Main*
begin

Running Example: *datatype-new iptrule-match* = *is-Src*: *Src* (*src-range*: *ipt-ipv4range*)

A discriminator *disc* tells whether a value is of a certain constructor. Example: *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* :: $(('a \Rightarrow bool) \times ('a \Rightarrow 'b)) \Rightarrow ('b \Rightarrow 'a) \Rightarrow bool$ **where**
  wf-disc-sel $(disc,\ sel)\ C \longleftrightarrow (\forall\, a.\ disc\ a \longrightarrow C\ (sel\ a) = a) \wedge (\forall\, a.\ (*disc\ (C\ a) \longrightarrow*)\ sel\ (C\ a) = a)$

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

end
theory *Negation-Type*
imports *Main*
begin

7 Negation Type

Only negated or non-negated literals

datatype *'a negation-type* = *Pos 'a* | *Neg 'a*

fun *getPos* :: *'a negation-type list* ⇒ *'a list* **where**
  getPos [] = [] |
  getPos ((*Pos x*)#*xs*) = *x*#(*getPos xs*) |
  getPos (-#*xs*) = *getPos xs*

fun *getNeg* :: *'a negation-type list* ⇒ *'a list* **where**
  getNeg [] = [] |
  getNeg ((*Neg x*)#*xs*) = *x*#(*getNeg xs*) |
  getNeg (-#*xs*) = *getNeg xs*

If there is *'a negation-type*, then apply a *map* only to *'a*. I.e. keep *Neg* and *Pos*

fun *NegPos-map* :: (*'a* ⇒ *'b*) ⇒ *'a negation-type list* ⇒ *'b negation-type list* **where**
  NegPos-map - [] = [] |
  NegPos-map f ((*Pos a*)#*as*) = (*Pos* (*f a*))#*NegPos-map f as* |
  NegPos-map f ((*Neg a*)#*as*) = (*Neg* (*f a*))#*NegPos-map f as*

Example

lemma *NegPos-map* (λx::*nat*. $x+1$) [*Pos 0*, *Neg 1*] = [*Pos 1*, *Neg 2*] **by** *eval*

lemma *getPos-NegPos-map-simp*: (*getPos* (*NegPos-map X* (*map Pos src*))) = *map X src*
  by(*induction src*) (*simp-all*)
lemma *getNeg-NegPos-map-simp*: (*getNeg* (*NegPos-map X* (*map Neg src*))) = *map X src*
  by(*induction src*) (*simp-all*)
lemma *getNeg-Pos-empty*: (*getNeg* (*NegPos-map X* (*map Pos src*))) = []
  by(*induction src*) (*simp-all*)
lemma *getNeg-Neg-empty*: (*getPos* (*NegPos-map X* (*map Neg src*))) = []
  by(*induction src*) (*simp-all*)
lemma *getPos-NegPos-map-simp2*: (*getPos* (*NegPos-map X src*)) = *map X* (*getPos src*)
  by(*induction src rule*: *getPos.induct*) (*simp-all*)
lemma *getNeg-NegPos-map-simp2*: (*getNeg* (*NegPos-map X src*)) = *map X* (*getNeg src*)
  by(*induction src rule*: *getPos.induct*) (*simp-all*)
lemma *getPos-id*: (*getPos* (*map Pos* (*getPos src*))) = *getPos src*
  by(*induction src rule*: *getPos.induct*) (*simp-all*)
lemma *getNeg-id*: (*getNeg* (*map Neg* (*getNeg src*))) = *getNeg src*
  by(*induction src rule*: *getNeg.induct*) (*simp-all*)
lemma *getPos-empty2*: (*getPos* (*map Neg src*)) = []
  by(*induction src*) (*simp-all*)
lemma *getNeg-empty2*: (*getNeg* (*map Pos src*)) = []
  by(*induction src*) (*simp-all*)

lemmas *NegPos-map-simps* = *getPos-NegPos-map-simp getNeg-NegPos-map-simp getNeg-Pos-empty getNeg-Neg-empty getPos-NegPos-map-simp2 getNeg-NegPos-map-simp2 getPos-id getNeg-id getPos-empty2 getNeg-empty2*

lemma *NegPos-map-append*: *NegPos-map C* (*as* @ *bs*) = *NegPos-map C as* @ *NegPos-map C bs*
  by(*induction as rule*: *getNeg.induct*) (*simp-all*)

lemma *getPos-set*: *Pos* $a \in$ *set* $x \longleftrightarrow a \in$ *set* (*getPos* x)
 apply(*induction x rule*: *getPos.induct*)
 apply(*auto*)
 done
lemma *getNeg-set*: *Neg* $a \in$ *set* $x \longleftrightarrow a \in$ *set* (*getNeg* x)
 apply(*induction x rule*: *getPos.induct*)
 apply(*auto*)
 done
lemma *getPosgetNeg-subset*: *set* $x \subseteq$ *set* $x' \longleftrightarrow$ *set* (*getPos* x) $\subseteq$ *set* (*getPos* x') $\wedge$ *set* (*getNeg* x) $\subseteq$ *set* (*getNeg* x')
  apply(*induction x rule*: *getPos.induct*)
  apply(*simp*)
  apply(*simp add*: *getPos-set*)
  apply(*rule iffI*)

apply(*simp-all add*: *getPos-set getNeg-set*)
done
lemma *set-Pos-getPos-subset*: *Pos ‘ set* (*getPos x*) ⊆ *set x*
apply(*induction x rule*: *getPos.induct*)
apply(*simp-all*)
apply *blast+*
done
lemma *set-Neg-getNeg-subset*: *Neg ‘ set* (*getNeg x*) ⊆ *set x*
apply(*induction x rule*: *getNeg.induct*)
apply(*simp-all*)
apply *blast+*
done
lemmas *NegPos-set* = *getPos-set getNeg-set getPosgetNeg-subset set-Pos-getPos-subset set-Neg-getNeg-subset*
hide-fact *getPos-set getNeg-set getPosgetNeg-subset set-Pos-getPos-subset set-Neg-getNeg-subset*

lemma *negation-type-forall-split*: (∀ *is*∈*set Ms*. *case is of Pos i* ⇒ *P i* | *Neg i* ⇒ *Q i*) ⟷ (∀ *i*∈*set* (*getPos Ms*). *P i*) ∧ (∀ *i*∈*set* (*getNeg Ms*). *Q i*)
apply(*rule*)
apply(*simp split*: *negation-type.split-asm*)
using *NegPos-set*(*1*) *NegPos-set*(*2*) **apply** *force*
apply(*simp split*: *negation-type.split*)
using *NegPos-set*(*1*) *NegPos-set*(*2*) **by** *fastforce*

fun *invert* :: *'a negation-type* ⇒ *'a negation-type* **where**
invert (*Pos x*) = *Neg x* |
invert (*Neg x*) = (*Pos x*)

end
theory *WordInterval-Lists*
imports *WordInterval*
../Common/Negation-Type
begin

7.1 WordInterval to List

A list of (*start*, *end*) tuples.

fun *br2l* :: *'a*::*len wordinterval* ⇒ (*'a*::*len word* × *'a*::*len word*) *list* **where**
br2l (*RangeUnion r1 r2*) = *br2l r1* @ *br2l r2* |
br2l (*WordInterval s e*) = (*if e* < *s then* [] *else* [(*s*,*e*)])

fun *l2br* :: (*'a*::*len word* × *'a*::*len word*) *list* ⇒ *'a*::*len wordinterval* **where**
l2br [] = *Empty-WordInterval* |
l2br [(*s*,*e*)] = (*WordInterval s e*) |
l2br ((*s*,*e*)#*rs*) = (*RangeUnion* (*WordInterval s e*) (*l2br rs*))

lemma *l2br-append*: *wordinterval-to-set* (*l2br* (*l1*@*l2*)) = *wordinterval-to-set*

(*l2br l1*) ∪ *wordinterval-to-set* (*l2br l2*)
apply(*induction l1 arbitrary*: *l2 rule*:*l2br.induct*)
apply(*simp-all*)
apply(*case-tac l2*)
apply(*simp-all*)
by *blast*

lemma *l2br-br2l*: *wordinterval-to-set* (*l2br* (*br2l r*)) = *wordinterval-to-set r*
by(*induction r*) (*simp-all add*: *l2br-append*)

lemma *l2br*: *wordinterval-to-set* (*l2br l*) = ($\bigcup$ (*i*,*j*) ∈ *set l*. {*i* .. *j*})
by(*induction l rule*: *l2br.induct*, *simp-all*)

definition *l-br-toset* :: (′*a*::*len word* × ′*a*::*len word*) *list* ⇒ (′*a*::*len word*) *set*
where
l-br-toset l ≡ $\bigcup$ (*i*,*j*) ∈ *set l*. {*i* .. *j*}
lemma *l-br-toset*: *l-br-toset l* = *wordinterval-to-set* (*l2br l*)
unfolding *l-br-toset-def*
apply(*induction l rule*: *l2br.induct*)
apply(*simp-all*)
done

definition *l2br-intersect* :: (′*a*::*len word* × ′*a*::*len word*) *list* ⇒ ′*a*::*len wordinterval* **where**
l2br-intersect = *foldl* (λ *acc* (*s*,*e*). *wordinterval-intersection* (*WordInterval s e*) *acc*) *wordinterval-UNIV*

lemma *l2br-intersect*: *wordinterval-to-set* (*l2br-intersect l*) = ($\bigcap$ (*i*,*j*) ∈ *set l*. {*i* .. *j*})
proof −
{ fix *U* — *wordinterval-UNIV* generalized
have *wordinterval-to-set* (*foldl* (λ*acc* (*s*, *e*). *wordinterval-intersection* (*WordInterval s e*) *acc*) *U l*) = (*wordinterval-to-set U*) ∩ ($\bigcap$(*i*, *j*)∈*set l*. {*i*..*j*})
apply(*induction l arbitrary*: *U*)
apply(*simp*)
by *force*
} thus *?thesis*
unfolding *l2br-intersect-def* **by** *simp*
qed

fun *l2br-negation-type-intersect* :: (′*a*::*len word* × ′*a*::*len word*) *negation-type list* ⇒ ′*a*::*len wordinterval* **where**

l2br-negation-type-intersect [] = *wordinterval-UNIV* |
l2br-negation-type-intersect ((*Pos* (*s*,*e*))#*ls*) = *wordinterval-intersection* (*WordInterval* *s* *e*) (*l2br-negation-type-intersect* *ls*) |
l2br-negation-type-intersect ((*Neg* (*s*,*e*))#*ls*) = *wordinterval-intersection* (*wordinterval-invert* (*WordInterval* *s* *e*)) (*l2br-negation-type-intersect* *ls*)

lemma *l2br-negation-type-intersect-alt*: *wordinterval-to-set* (*l2br-negation-type-intersect* *l*) =
wordinterval-to-set (*wordinterval-setminus* (*l2br-intersect* (*getPos* *l*)) (*l2br* (*getNeg* *l*)))
apply(*simp add*: *l2br-intersect l2br*)
apply(*induction l rule* :*l2br-negation-type-intersect.induct*)
apply(*simp-all*)
apply(*fast*)+
done

lemma *l2br-negation-type-intersect*: *wordinterval-to-set* (*l2br-negation-type-intersect* *l*) =
($\bigcap$ (*i*,*j*) ∈ *set* (*getPos* *l*). {*i* .. *j*}) − ($\bigcup$ (*i*,*j*) ∈ *set* (*getNeg* *l*). {*i* .. *j*})
by(*simp add*: *l2br-negation-type-intersect-alt l2br-intersect l2br*)

fun *l2br-negation-type-union* :: ('*a*::*len word* × '*a*::*len word*) *negation-type list* ⇒ '*a*::*len wordinterval* **where**
l2br-negation-type-union [] = *Empty-WordInterval* |
l2br-negation-type-union ((*Pos* (*s*,*e*))#*ls*) = *wordinterval-union* (*WordInterval* *s* *e*) (*l2br-negation-type-union* *ls*) |
l2br-negation-type-union ((*Neg* (*s*,*e*))#*ls*) = *wordinterval-union* (*wordinterval-invert* (*WordInterval* *s* *e*)) (*l2br-negation-type-union* *ls*)

lemma *l2br-negation-type-union*: *wordinterval-to-set* (*l2br-negation-type-union* *l*) =
($\bigcup$ (*i*,*j*) ∈ *set* (*getPos* *l*). {*i* .. *j*}) ∪ ($\bigcup$ (*i*,*j*) ∈ *set* (*getNeg* *l*). − {*i* .. *j*})
apply(*simp add*: *l2br*)
apply(*induction l rule*: *l2br-negation-type-union.induct*)
apply(*simp-all*)
apply *fast*+
done
end
theory *IpAddresses*
imports ../*Bitmagic*/*IPv4Addr*
../*Bitmagic*/*Numberwang-Ln*
../*Bitmagic*/*CIDRSplit*
../*Bitmagic*/*WordInterval-Lists*
begin

8 IPv4 Addresses

— Misc

lemma *ipv4range-set-from-bitmask* (*ipv4addr-of-dotdecimal* (*0*, *0*, *0*, *0*)) *33* = {*0*}
apply(*simp add*: *ipv4addr-of-dotdecimal.simps ipv4addr-of-nat-def*)
apply(*simp add*: *ipv4range-set-from-bitmask-def*)
apply(*simp add*: *ipv4range-set-from-netmask-def*)
done

8.1 IPv4 Addresses in CIDR Notation

fun *ipv4cidr-to-interval* :: (*ipv4addr* × *nat*) ⇒ (*ipv4addr* × *ipv4addr*) **where**
ipv4cidr-to-interval (*pre*, *len*) = (
let netmask = (*mask len*) << (*32* − *len*);
network-prefix = (*pre AND netmask*)
in (*network-prefix*, *network-prefix OR* (*NOT netmask*))
)
lemma *ipv4cidr-to-interval*: *ipv4cidr-to-interval* (*base*, *len*) = (*s*,*e*) ⟹ *ipv4range-set-from-bitmask base len* = {*s* .. *e*}
apply(*simp add*: *Let-def*)
apply(*subst ipv4range-set-from-bitmask-alt*)
apply(*subst*(*asm*) *NOT-mask-len32*)
by (*metis NOT-mask-len32 ipv4range-set-from-bitmask-alt ipv4range-set-from-bitmask-alt1 ipv4range-set-from-netmask-def*)
declare *ipv4cidr-to-interval.simps*[*simp del*]

fun *ipv4cidr-conjunct* :: (*ipv4addr* × *nat*) ⇒ (*ipv4addr* × *nat*) ⇒ (*ipv4addr* × *nat*) *option* **where**
ipv4cidr-conjunct (*base1*, *m1*) (*base2*, *m2*) = (*if ipv4range-set-from-bitmask base1 m1* ∩ *ipv4range-set-from-bitmask base2 m2* = {}
then
None
else if
ipv4range-set-from-bitmask base1 m1 ⊆ *ipv4range-set-from-bitmask base2 m2*
then
Some (*base1*, *m1*)
else
Some (*base2*, *m2*)
)

lemma *ipv4cidr-conjunct-correct*: (*case ipv4cidr-conjunct* (*b1*, *m1*) (*b2*, *m2*) *of Some* (*bx*, *mx*) ⇒ *ipv4range-set-from-bitmask bx mx* | *None* ⇒ {}) =
(*ipv4range-set-from-bitmask b1 m1*) ∩ (*ipv4range-set-from-bitmask b2 m2*)
apply(*simp split*: *split-if-asm*)
using *ipv4range-bitmask-intersect* **by** *fast*
declare *ipv4cidr-conjunct.simps*[*simp del*]

definition *ipv4-cidr-tuple-to-interval* :: (*ipv4addr* × *nat*) ⇒ *32 wordinterval*

where
  ipv4-cidr-tuple-to-interval iprng = ipv4range-range (ipv4cidr-to-interval iprng)

lemma *ipv4range-to-set-ipv4-cidr-tuple-to-interval*: *ipv4range-to-set (ipv4-cidr-tuple-to-interval (b, m)) = ipv4range-set-from-bitmask b m*
  unfolding *ipv4-cidr-tuple-to-interval-def*
  apply(*cases ipv4cidr-to-interval (b, m)*)
  using *ipv4cidr-to-interval ipv4range-range-set-eq* **by** *presburger*

lemma [*code-unfold*]:
ipv4cidr-conjunct ips1 ips2 = (if ipv4range-empty (ipv4range-intersection (ipv4-cidr-tuple-to-interval ips1) (ipv4-cidr-tuple-to-interval ips2))
    then
     None
    else if
    ipv4range-subset (ipv4-cidr-tuple-to-interval ips1) (ipv4-cidr-tuple-to-interval ips2)
    then
     Some ips1
    else
     Some ips2
)
apply(*simp*)
apply(*cases ips1, cases ips2, rename-tac b1 m1 b2 m2, simp*)
apply(*safe*)
 apply(*simp-all add: ipv4range-to-set-ipv4-cidr-tuple-to-interval ipv4cidr-conjunct.simps split:split-if-asm*)
 apply *fast+*
done
value *ipv4cidr-conjunct (0,0) (8,1)*

definition *ipv4cidr-union-set* :: *(ipv4addr × nat) set ⇒ ipv4addr set* **where**
  ipv4cidr-union-set ips ≡ ⋃(base, len) ∈ ips. ipv4range-set-from-bitmask base len

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

datatype *ipt-ipv4range = Ip4Addr nat × nat × nat × nat*
      | *Ip4AddrNetmask nat × nat × nat × nat nat* — addr/xx

fun *ipv4s-to-set* :: *ipt-ipv4range ⇒ ipv4addr set* **where**
 ipv4s-to-set (Ip4AddrNetmask base m) = ipv4range-set-from-bitmask (ipv4addr-of-dotdecimal base) m |
  ipv4s-to-set (Ip4Addr ip) = { ipv4addr-of-dotdecimal ip }

ipv4s-to-set cannot represent an empty set.

lemma *ipv4s-to-set-nonempty*: *ipv4s-to-set ip ≠ {}*

apply(*cases ip*)
apply(*simp*)
apply(*simp add*: *ipv4range-set-from-bitmask-alt*)
apply(*simp add*: *bitmagic-zeroLast-leq-or1Last*)
done

maybe this is necessary as code equation?

lemma *element-ipv4s-to-set*[*code-unfold*]: $addr \in ipv4s\text{-}to\text{-}set\ X = ($
$case\ X\ of\ (Ip4AddrNetmask\ pre\ len) \Rightarrow ((ipv4addr\text{-}of\text{-}dotdecimal\ pre)\ AND$
$((mask\ len) << (32 - len))) \leq addr \wedge addr \leq (ipv4addr\text{-}of\text{-}dotdecimal\ pre)\ OR$
$(mask\ (32 - len))$
$|\ Ip4Addr\ ip \Rightarrow (addr = (ipv4addr\text{-}of\text{-}dotdecimal\ ip))\ )$
apply(*cases X*)
apply(*simp*)
apply(*simp add*: *ipv4range-set-from-bitmask-alt*)
done

IPv4 address ranges to (*start*, *end*) notation

fun *ipt-ipv4range-to-interval* :: *ipt-ipv4range* ⇒ (*ipv4addr* × *ipv4addr*) **where**
ipt-ipv4range-to-interval (*Ip4Addr addr*) = (*ipv4addr-of-dotdecimal addr*, *ipv4addr-of-dotdecimal addr*) |
ipt-ipv4range-to-interval (*Ip4AddrNetmask pre len*) = *ipv4cidr-to-interval* ((*ipv4addr-of-dotdecimal pre*), *len*)

lemma *ipt-ipv4range-to-interval*: *ipt-ipv4range-to-interval ip* = (*s*,*e*) ⟹ {*s* .. *e*} = *ipv4s-to-set ip*
by(*cases ip*) (*auto simp add*: *ipv4cidr-to-interval*)

A list of IPv4 address ranges to a *32 wordinterval*. The nice thing is: the usual set operations are defined on this type. We can use the existing function *l2br-intersect* if we want the intersection of the supplied list

lemma *wordinterval-to-set* (*l2br-intersect* (*map ipt-ipv4range-to-interval ips*)) = (⋂ *ip* ∈ *set ips*. *ipv4s-to-set ip*)
apply(*simp add*: *l2br-intersect*)
using *ipt-ipv4range-to-interval* **by** *blast*

We can use *l2br* if we want the union of the supplied list

lemma *wordinterval-to-set* (*l2br* (*map ipt-ipv4range-to-interval ips*)) = (⋃ *ip* ∈ *set ips*. *ipv4s-to-set ip*)
apply(*simp add*: *l2br*)
using *ipt-ipv4range-to-interval* **by** *blast*

A list of (negated) IPv4 address to a *32 wordinterval*.

definition *ipt-ipv4range-negation-type-to-br-intersect* :: *ipt-ipv4range negation-type list* ⇒ *32 wordinterval* **where**
ipt-ipv4range-negation-type-to-br-intersect l = *l2br-negation-type-intersect* (*NegPos-map ipt-ipv4range-to-interval l*)

lemma *ipt-ipv4range-negation-type-to-br-intersect*: *wordinterval-to-set* (*ipt-ipv4range-negation-type-to-br-inte*
l) =
  ($\bigcap$ *ip* $\in$ *set* (*getPos l*). *ipv4s-to-set ip*) $-$ ($\bigcup$ *ip* $\in$ *set* (*getNeg l*). *ipv4s-to-set*
ip)
apply(*simp add*: *ipt-ipv4range-negation-type-to-br-intersect-def l2br-negation-type-intersect*
NegPos-map-simps)
using *ipt-ipv4range-to-interval* **by** *blast*

The *32 wordinterval* can be translated back into a list of IP ranges. If a list of intervals is enough, we can use *br2l*. If we need it in *ipt-ipv4range*, we can use this function.

definition *br-2-cidr-ipt-ipv4range-list* :: *32 wordinterval* $\Rightarrow$ *ipt-ipv4range list*
where
br-2-cidr-ipt-ipv4range-list r = *map* (λ (*base*, *len*). *Ip4AddrNetmask* (*dotdecimal-of-ipv4addr*
base) *len*) (*ipv4range-split r*)

lemma *br-2-cidr-ipt-ipv4range-list*: ($\bigcup$ *ip* $\in$ *set* (*br-2-cidr-ipt-ipv4range-list r*).
ipv4s-to-set ip) = *wordinterval-to-set r*
proof $-$
have $\bigwedge$*a*. *ipv4s-to-set* (*case a of* (*base*, *x*) $\Rightarrow$ *Ip4AddrNetmask* (*dotdecimal-of-ipv4addr*
base) *x*) = (*case a of* (*x*, *xa*) $\Rightarrow$ *ipv4range-set-from-bitmask x xa*)
by(*clarsimp simp add*: *ipv4addr-of-dotdecimal-dotdecimal-of-ipv4addr*)
hence ($\bigcup$ *ip* $\in$ *set* (*br-2-cidr-ipt-ipv4range-list r*). *ipv4s-to-set ip*) = $\bigcup$((λ(*x*,
y). *ipv4range-set-from-bitmask x y*) ‘ *set* (*ipv4range-split r*))
unfolding *br-2-cidr-ipt-ipv4range-list-def* **by**(*simp*)
thus *?thesis*
using *ipv4range-split-bitmask* **by** *presburger*
qed

For example, this allows the following transformation

definition *ipt-ipv4range-compress* :: *ipt-ipv4range negation-type list* $\Rightarrow$ *ipt-ipv4range*
list **where**
ipt-ipv4range-compress = *br-2-cidr-ipt-ipv4range-list* $\circ$ *ipt-ipv4range-negation-type-to-br-intersect*

lemma *ipt-ipv4range-compress*: ($\bigcup$ *ip* $\in$ *set* (*ipt-ipv4range-compress l*). *ipv4s-to-set*
ip) =
  ($\bigcap$ *ip* $\in$ *set* (*getPos l*). *ipv4s-to-set ip*) $-$ ($\bigcup$ *ip* $\in$ *set* (*getNeg l*). *ipv4s-to-set*
ip)
by (*metis br-2-cidr-ipt-ipv4range-list comp-apply ipt-ipv4range-compress-def*
ipt-ipv4range-negation-type-to-br-intersect)

end
theory *Iface*
imports *String ../Common/Negation-Type*
begin

9 Network Interfaces

datatype *iface* = *Iface* (*iface-sel*: *string*) — no negation supported, but wildcards

definition *ifaceAny* :: *iface* **where**
ifaceAny ≡ *Iface* ''+''*If the interface name ends in a "+", then any interface which begins with this name will match. (man iptables)*
Here is how iptables handles this wildcard on my system. A packet for the loopback interface `lo` *is matched by the following expressions*

- *lo*
- *lo+*
- *l+*
- *+*

It is not matched by the following expressions

- *lo++*
- *lo+++*
- *lo1+*
- *lo1*

By the way: `Warning: weird characters in interface ' ' ('/' and ' ' are not allowed by the kernel).` **context**
begin

9.1 Helpers for the interface name (*string*)

argument 1: interface as in firewall rule - Wildcard support argument 2: interface a packet came from - No wildcard support

private fun *internal-iface-name-match* :: *string* ⇒ *string* ⇒ *bool* **where**
internal-iface-name-match [] [] ⟷ *True* |
internal-iface-name-match (*i*#*is*) [] ⟷ (*i* = *CHR* ''+'' ∧ *is* = []) |
internal-iface-name-match [] (-#-) ⟷ *False* |
internal-iface-name-match (*i*#*is*) (*p-i*#*p-is*) ⟷ (*if* (*i* = *CHR* ''+'' ∧ *is* = []) *then True else* (
(*p-i* = *i*) ∧ *internal-iface-name-match is p-is*
))

private fun *iface-name-is-wildcard* :: *string* ⇒ *bool* **where**
iface-name-is-wildcard [] ⟷ *False* |
iface-name-is-wildcard [*s*] ⟷ *s* = *CHR* ''+'' |
iface-name-is-wildcard (-#*ss*) ⟷ *iface-name-is-wildcard ss*
private lemma *iface-name-is-wildcard-alt*: *iface-name-is-wildcard eth* ⟷ *eth* ≠ [] ∧ *last eth* = *CHR* ''+''

apply(*induction eth rule*: *iface-name-is-wildcard.induct*)
apply(*simp-all*)
done

private lemma *iface-name-is-wildcard-alt'*: *iface-name-is-wildcard eth* $\longleftrightarrow$ *eth* $\neq$ [] $\wedge$ *hd* (*rev eth*) = *CHR* ''+''
apply(*simp add*: *iface-name-is-wildcard-alt*)
using *hd-rev* **by** *fastforce*

private lemma *iface-name-is-wildcard-fst*: *iface-name-is-wildcard* (*i* # *is*) $\Longrightarrow$ *is* $\neq$ [] $\Longrightarrow$ *iface-name-is-wildcard is*
by(*simp add*: *iface-name-is-wildcard-alt*)

private fun *internal-iface-name-to-set* :: *string* $\Rightarrow$ *string set* **where**
internal-iface-name-to-set i = (*if* $\neg$ *iface-name-is-wildcard i*
then
{*i*}
else
{(*butlast i*)@*cs* | *cs*. *True*})

private lemma {(*butlast i*)@*cs* | *cs*. *True*} = (λ*s*. (*butlast i*)@*s*) ` (*UNIV*::*string set*) **by** *fastforce*

private lemma *internal-iface-name-to-set*: *internal-iface-name-match i p-iface* $\longleftrightarrow$ *p-iface* $\in$ *internal-iface-name-to-set i*
apply(*induction i p-iface rule*: *internal-iface-name-match.induct*)
apply(*simp-all*)
apply(*safe*)
apply(*simp-all add*: *iface-name-is-wildcard-fst*)
apply (*metis* (*full-types*) *iface-name-is-wildcard.simps*(*3*) *list.exhaust*)
by (*metis append-butlast-last-id*)

private lemma *internal-iface-name-to-set2*: *internal-iface-name-to-set ifce* = {*i*. *internal-iface-name-match ifce i*}
by (*simp add*: *internal-iface-name-to-set*)

private lemma *internal-iface-name-match-refl*: *internal-iface-name-match i i*
proof −
{ **fix** *i j*
have *i*=*j* $\Longrightarrow$ *internal-iface-name-match i j*
by(*induction i j rule*: *internal-iface-name-match.induct*)(*simp-all*)
} **thus** *?thesis* **by** *simp*
qed

9.2 Matching

fun *match-iface* :: *iface* $\Rightarrow$ *string* $\Rightarrow$ *bool* **where**
match-iface (*Iface i*) *p-iface* $\longleftrightarrow$ *internal-iface-name-match i p-iface*

— Examples
lemma *match-iface* (*Iface* ''lo'') ''lo''
match-iface (*Iface* ''lo+'') ''lo''
match-iface (*Iface* ''l+'') ''lo''

$match\text{-}iface\ (Iface\ ''+'')\quad ''lo''$
$\neg\ match\text{-}iface\ (Iface\ ''lo{+}{+}'')\ ''lo''$
$\neg\ match\text{-}iface\ (Iface\ ''lo{+}{+}{+}'')\ ''lo''$
$\neg\ match\text{-}iface\ (Iface\ ''lo1{+}'')\ ''lo''$
$\neg\ match\text{-}iface\ (Iface\ ''lo1'')\ ''lo''$
$match\text{-}iface\ (Iface\ ''+'')\quad ''eth0''$
$match\text{-}iface\ (Iface\ ''+'')\quad ''eth0''$
$match\text{-}iface\ (Iface\ ''eth{+}'')\ ''eth0''$
$\neg\ match\text{-}iface\ (Iface\ ''lo{+}'')\ ''eth0''$
$match\text{-}iface\ (Iface\ ''lo{+}'')\ ''loX''$
$\neg\ match\text{-}iface\ (Iface\ '''')\quad ''loX''$

lemma *match-ifaceAny*: *match-iface ifaceAny i*
by(*cases i*, *simp-all add*: *ifaceAny-def*)
lemma *match-IfaceFalse*: $\neg(\exists\ IfaceFalse.\ (\forall\, i.\ \neg\ match\text{-}iface\ IfaceFalse\ i))$
apply(*simp*)
apply(*intro allI*, *rename-tac IfaceFalse*)
apply(*case-tac IfaceFalse*, *rename-tac name*)
apply(*rule-tac x=name* **in** *exI*)
by(*simp add*: *internal-iface-name-match-refl*)

— *match-iface* explained by the individual cases
lemma *match-iface-case-nowildcard*: $\neg$ *iface-name-is-wildcard i* $\Longrightarrow$ *match-iface* (*Iface i*) *p-i* $\longleftrightarrow$ *i* = *p-i*
apply(*simp*)
apply(*induction i p-i rule*: *internal-iface-name-match.induct*)
apply(*auto simp add*: *iface-name-is-wildcard-alt split*: *split-if-asm*)
done
lemma *match-iface-case-wildcard-prefix*:
iface-name-is-wildcard i $\Longrightarrow$ *match-iface* (*Iface i*) *p-i* $\longleftrightarrow$ *butlast i* = *take* (*length i* $-$ *1*) *p-i*
apply(*simp*)
apply(*induction i p-i rule*: *internal-iface-name-match.induct*)
apply(*simp-all*)
apply(*simp add*: *iface-name-is-wildcard-alt split*: *split-if-asm*)
apply(*intro conjI*)
apply(*simp add*: *iface-name-is-wildcard-alt split*: *split-if-asm*)
apply(*intro impI*)
apply(*simp add*: *iface-name-is-wildcard-fst*)
by (*metis One-nat-def length-0-conv list.sel*(*1*) *list.sel*(*3*) *take-Cons'*)
lemma *match-iface-case-wildcard-length*: *iface-name-is-wildcard i* $\Longrightarrow$ *match-iface* (*Iface i*) *p-i* $\Longrightarrow$ *length p-i* $\geq$ (*length i* $-$ *1*)
apply(*simp*)
apply(*induction i p-i rule*: *internal-iface-name-match.induct*)
apply(*simp-all*)
apply(*simp add*: *iface-name-is-wildcard-alt split*: *split-if-asm*)
done

corollary *match-iface-case-wildcard*:
iface-name-is-wildcard i $\Longrightarrow$ *match-iface* (*Iface i*) *p-i* $\longleftrightarrow$ *butlast i* = *take* (*length i* − *1*) *p-i* ∧ *length p-i* ≥ (*length i* − *1*)
using *match-iface-case-wildcard-length match-iface-case-wildcard-prefix* **by** *blast*

lemma *match-iface-set*: *match-iface* (*Iface i*) *p-iface* $\longleftrightarrow$ *p-iface* ∈ *internal-iface-name-to-set i*
using *internal-iface-name-to-set* **by** *simp*

private definition *internal-iface-name-wildcard-longest* :: *string* ⇒ *string* ⇒ *string option* **where**
internal-iface-name-wildcard-longest i1 i2 = (
if
take (*min* (*length i1* − *1*) (*length i2* − *1*)) *i1* = *take* (*min* (*length i1* − *1*) (*length i2* − *1*)) *i2*
then
Some (*if length i1* ≤ *length i2 then i2 else i1*)
else
None)
private lemma *internal-iface-name-wildcard-longest* ''*eth+*'' ''*eth3+*'' = *Some* ''*eth3+*'' **by** *eval*
private lemma *internal-iface-name-wildcard-longest* ''*eth+*'' ''*e+*'' = *Some* ''*eth+*'' **by** *eval*
private lemma *internal-iface-name-wildcard-longest* ''*eth+*'' ''*lo*'' = *None* **by** *eval*

private lemma *internal-iface-name-wildcard-longest-commute*: *iface-name-is-wildcard i1* $\Longrightarrow$ *iface-name-is-wildcard i2* $\Longrightarrow$
internal-iface-name-wildcard-longest i1 i2 = *internal-iface-name-wildcard-longest i2 i1*
apply(*simp add*: *internal-iface-name-wildcard-longest-def*)
apply(*safe*)
apply(*simp-all add*: *iface-name-is-wildcard-alt*)
apply (*metis One-nat-def append-butlast-last-id butlast-conv-take*)
by (*metis min.commute*)+
private lemma *internal-iface-name-wildcard-longest-refl*: *internal-iface-name-wildcard-longest i i* = *Some i*
by(*simp add*: *internal-iface-name-wildcard-longest-def*)

private lemma *internal-iface-name-wildcard-longest-correct*: *iface-name-is-wildcard i1* $\Longrightarrow$ *iface-name-is-wildcard i2* $\Longrightarrow$
match-iface (*Iface i1*) *p-i* ∧ *match-iface* (*Iface i2*) *p-i* $\longleftrightarrow$
(*case internal-iface-name-wildcard-longest i1 i2 of None* ⇒ *False* | *Some x* ⇒ *match-iface* (*Iface x*) *p-i*)
proof −
assume *assm1*: *iface-name-is-wildcard i1*

and *assm2*: *iface-name-is-wildcard i2*
{ assume *assm3*: *internal-iface-name-wildcard-longest i1 i2* = *None*
have ¬ (*internal-iface-name-match i1 p-i* ∧ *internal-iface-name-match i2 p-i*)
proof −
from *match-iface-case-wildcard-prefix*[*OF assm1*] **have** *1*:
internal-iface-name-match i1 p-i = (*take* (*length i1* − *1*) *i1* = *take* (*length i1* − *1*) *p-i*) **by**(*simp add*: *butlast-conv-take*)
from *match-iface-case-wildcard-prefix*[*OF assm2*] **have** *2*:
internal-iface-name-match i2 p-i = (*take* (*length i2* − *1*) *i2* = *take* (*length i2* − *1*) *p-i*) **by**(*simp add*: *butlast-conv-take*)
from *assm3* **have** *3*: *take* (*min* (*length i1* − *1*) (*length i2* − *1*)) *i1* ≠ *take* (*min* (*length i1* − *1*) (*length i2* − *1*)) *i2*
by(*simp add*: *internal-iface-name-wildcard-longest-def split*: *split-if-asm*)
from *3* **show** *?thesis* **using** *1 2 min.commute take-take* **by** *metis*
qed
} note *internal-iface-name-wildcard-longest-correct-None=this*

{ fix *X*
assume *assm3*: *internal-iface-name-wildcard-longest i1 i2* = *Some X*
have (*internal-iface-name-match i1 p-i* ∧ *internal-iface-name-match i2 p-i*) ⟷ *internal-iface-name-match X p-i*
proof −
from *assm3* **have** *assm3'*: *take* (*min* (*length i1* − *1*) (*length i2* − *1*)) *i1* = *take* (*min* (*length i1* − *1*) (*length i2* − *1*)) *i2*
unfolding *internal-iface-name-wildcard-longest-def* **by**(*simp split*: *split-if-asm*)

{ fix *i1 i2*
assume *iw1*: *iface-name-is-wildcard i1* **and** *iw2*: *iface-name-is-wildcard i2* **and** *len*: *length i1* ≤ *length i2* **and**
take-i1i2: *take* (*length i1* − *1*) *i1* = *take* (*length i1* − *1*) *i2*
from *len* **have** *len'*: *length i1* − *1* ≤ *length i2* − *1* **by** *fastforce*
{ fix *x::string*
from *len'* **have** *take* (*length i1* − *1*) *x* = *take* (*length i1* − *1*) (*take* (*length i2* − *1*) *x*) **by**(*simp add*: *min-def*)
} note *takei1=this*

{ fix *m::nat* **and** *n::nat* **and** *a::string* **and** *b c*
have *m* ≤ *n* ⟹ *take n a* = *take n b* ⟹ *take m a* = *take m c* ⟹ *take m c* = *take m b* **by** (*metis min-absorb1 take-take*)
} note *takesmaller=this*

from *match-iface-case-wildcard-prefix*[*OF iw1*, *simplified*] **have** *1*:
internal-iface-name-match i1 p-i ⟷ *take* (*length i1* − *1*) *i1* = *take* (*length i1* − *1*) *p-i* **by**(*simp add*: *butlast-conv-take*)
also have ... ⟷ *take* (*length i1* − *1*) (*take* (*length i2* − *1*) *i1*) = *take* (*length i1* − *1*) (*take* (*length i2* − *1*) *p-i*) **using** *takei1* **by** *simp*
finally have *internal-iface-name-match i1 p-i* = (*take* (*length i1* − *1*)

$(take\ (length\ i2\ -\ 1)\ i1) = take\ (length\ i1\ -\ 1)\ (take\ (length\ i2\ -\ 1)\ p\text{-}i))$.
from *match-iface-case-wildcard-prefix*[*OF iw2*, *simplified*] **have** *2*:
internal-iface-name-match i2 p-i $\longleftrightarrow$ *take* (*length i2* − *1*) *i2* = *take* (*length i2* − *1*) *p-i* **by**(*simp add*: *butlast-conv-take*)

have *internal-iface-name-match i2 p-i* $\Longrightarrow$ *internal-iface-name-match i1 p-i*
unfolding *1 2*
apply(*rule takesmaller*[*of* (*length i1* − *1*) (*length i2* − *1*) *i2 p-i*])
using *len′* **apply** (*simp*)
apply *simp*
using *take-i1i2* **apply** *simp*
done
} **note** *longer-iface-imp-shorter=this*

show *?thesis*
proof(*cases length i1* ≤ *length i2*)
case *True*
with *assm3* **have** *X* = *i2* **unfolding** *internal-iface-name-wildcard-longest-def* **by**(*simp split*: *split-if-asm*)
from *True assm3′* **have** *take-i1i2*: *take* (*length i1* − *1*) *i1* = *take* (*length i1* − *1*) *i2* **by** *linarith*
from *longer-iface-imp-shorter*[*OF assm1 assm2 True take-i1i2*] ‹*X* = *i2*›
show (*internal-iface-name-match i1 p-i* ∧ *internal-iface-name-match i2 p-i*) $\longleftrightarrow$ *internal-iface-name-match X p-i* **by** *fastforce*
next
case *False*
with *assm3* **have** *X* = *i1* **unfolding** *internal-iface-name-wildcard-longest-def* **by**(*simp split*: *split-if-asm*)
from *False assm3′* **have** *take-i1i2*: *take* (*length i2* − *1*) *i2* = *take* (*length i2* − *1*) *i1* **by** (*metis min-def min-diff*)
from *longer-iface-imp-shorter*[*OF assm2 assm1 - take-i1i2*] *False* ‹*X* = *i1*›
show (*internal-iface-name-match i1 p-i* ∧ *internal-iface-name-match i2 p-i*) $\longleftrightarrow$ *internal-iface-name-match X p-i* **by** *auto*
qed
qed
} **note** *internal-iface-name-wildcard-longest-correct-Some=this*

from *internal-iface-name-wildcard-longest-correct-None internal-iface-name-wildcard-longest-correct-Some* **show** *?thesis*
by(*simp split*:*option.split*)
qed

fun *iface-conjunct* :: *iface* ⇒ *iface* ⇒ *iface option* **where**
iface-conjunct (*Iface i1*) (*Iface i2*) = (*case* (*iface-name-is-wildcard i1*, *iface-name-is-wildcard i2*) *of*
(*True*, *True*) ⇒ *map-option Iface* (*internal-iface-name-wildcard-longest i1 i2*) |

(*True*, *False*) ⇒ (*if match-iface* (*Iface i1*) *i2 then Some* (*Iface i2*) *else None*) |
(*False*, *True*) ⇒ (*if match-iface* (*Iface i2*) *i1 then Some* (*Iface i1*) *else None*) |
(*False*, *False*) ⇒ (*if i1* = *i2 then Some* (*Iface i1*) *else None*))

lemma *iface-conjunct*: *match-iface i1 p-i* ∧ *match-iface i2 p-i* ⟷
(*case iface-conjunct i1 i2 of None* ⇒ *False* | *Some x* ⇒ *match-iface x p-i*)
apply(*cases i1*, *cases i2*, *rename-tac i1name i2name*)
apply(*simp split*: *bool.split option.split*)

apply(*auto simp*: *internal-iface-name-wildcard-longest-refl dest*: *internal-iface-name-wildcard-longest-corre*
apply (*metis match-iface.simps match-iface-case-nowildcard*)+
done

lemma *match-iface-refl*: *match-iface* (*Iface x*) *x* **by** (*simp add*: *internal-iface-name-match-refl*)

private definition *internal-iface-name-subset* :: *string* ⇒ *string* ⇒ *bool* **where**
internal-iface-name-subset i1 i2 = (*case* (*iface-name-is-wildcard i1*, *iface-name-is-wildcard i2*) *of*
(*True*, *True*) ⇒ *length i1* ≥ *length i2* ∧ *take* ((*length i2*) − *1*) *i1* = *butlast i2* |
(*True*, *False*) ⇒ *False* |
(*False*, *True*) ⇒ *take* (*length i2* − *1*) *i1* = *butlast i2* |
(*False*, *False*) ⇒ *i1* = *i2*
)

private lemma *hlp1*: {*x*. ∃ *cs*. *x* = *i1* @ *cs*} ⊆ {*x*. ∃ *cs*. *x* = *i2* @ *cs*} ⟹ *length i2* ≤ *length i1*
apply(*simp add*: *Set.Collect-mono-iff*)
by *force*
private lemma *hlp2*: {*x*. ∃ *cs*. *x* = *i1* @ *cs*} ⊆ {*x*. ∃ *cs*. *x* = *i2* @ *cs*} ⟹ *take* (*length i2*) *i1* = *i2*
apply(*simp add*: *Set.Collect-mono-iff*)
by *force*

private lemma *internal-iface-name-subset*: *internal-iface-name-subset i1 i2* ⟷
{*i*. *internal-iface-name-match i1 i*} ⊆ {*i*. *internal-iface-name-match i2 i*}
unfolding *internal-iface-name-subset-def*
apply(*case-tac iface-name-is-wildcard i1*)
apply(*case-tac* [!] *iface-name-is-wildcard i2*)
apply(*simp-all*)
defer

using *internal-iface-name-match-refl match-iface-case-nowildcard* **apply** *fastforce*
using *match-iface-case-nowildcard match-iface-case-wildcard-prefix* **apply** *force*
using *match-iface-case-nowildcard* **apply** *force*
apply(*rule*)
apply(*clarify*, *rename-tac x*)
apply(*drule-tac p-i=x* **in** *match-iface-case-wildcard-prefix*)+
apply(*simp*)
apply (*smt One-nat-def append-take-drop-id butlast-conv-take cancel-comm-monoid-add-class.diff-cancel diff-commute diff-diff-cancel diff-is-0-eq drop-take length-butlast take-append*)
apply(*subst*(*asm*) *internal-iface-name-to-set2*[*symmetric*])+
apply(*simp add*: *internal-iface-name-to-set*)
apply(*safe*)
apply(*drule hlp1*)
apply(*simp*)
apply (*metis One-nat-def Suc-pred diff-Suc-eq-diff-pred diff-is-0-eq iface-name-is-wildcard.simps*(*1*) *length-greater-0-conv*)
apply(*drule hlp2*)
apply(*simp*)
by (*metis One-nat-def butlast-conv-take length-butlast length-take take-take*)

definition *iface-subset* :: *iface* ⇒ *iface* ⇒ *bool* **where**
iface-subset i1 i2 ⟷ *internal-iface-name-subset* (*iface-sel i1*) (*iface-sel i2*)

lemma *iface-subset*: *iface-subset i1 i2* ⟷ {*i*. *match-iface i1 i*} ⊆ {*i*. *match-iface i2 i*}
unfolding *iface-subset-def*
apply(*cases i1*, *cases i2*)
by(*simp add*: *internal-iface-name-subset*)

definition *iface-is-wildcard* :: *iface* ⇒ *bool* **where**
iface-is-wildcard ifce ≡ *iface-name-is-wildcard* (*iface-sel ifce*)

declare *match-iface.simps*[*simp del*]
declare *iface-name-is-wildcard.simps*[*simp del*]
end

end
theory *Protocol*
imports ../*Common*/*Negation-Type*
begin

datatype *primitive-protocol* = *TCP* | *UDP* | *ICMP*

datatype *protocol* = *ProtoAny* | *Proto primitive-protocol*

fun *match-proto* :: *protocol* ⇒ *primitive-protocol* ⇒ *bool* **where**
match-proto ProtoAny - ⟷ *True* |
match-proto (*Proto* (*p*)) *p-p* ⟷ *p-p* = *p*

fun *simple-proto-conjunct* :: *protocol* ⇒ *protocol* ⇒ *protocol option* **where**
simple-proto-conjunct ProtoAny proto = *Some proto* |
simple-proto-conjunct proto ProtoAny = *Some proto* |
simple-proto-conjunct (*Proto p1*) (*Proto p2*) = (*if p1* = *p2 then Some* (*Proto p1*) *else None*)

lemma *simple-proto-conjunct-correct*: *match-proto p1 pkt* ∧ *match-proto p2 pkt* ⟷
(*case simple-proto-conjunct p1 p2 of None* ⇒ *False* | *Some proto* ⇒ *match-proto proto pkt*)
apply(*cases p1*)
apply(*simp-all*)
apply(*rename-tac pp1*)
apply(*cases p2*)
apply(*simp-all*)
done

end
theory *Ports*
imports *String*
~~/*src*/*HOL*/*Word*/*Word*
../*Bitmagic*/*WordInterval-Lists*
begin

10 Ports (layer 4)

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

list of (start, end) port ranges

type-synonym *ipt-ports* = (*16 word* × *16 word*) *list*

fun *ports-to-set* :: *ipt-ports* ⇒ (*16 word*) *set* **where**
ports-to-set [] = {} |
ports-to-set ((*s*,*e*)#*ps*) = {*s*..*e*} ∪ *ports-to-set ps*

lemma *ports-to-set*: *ports-to-set pts* = ⋃ {{*s*..*e*} | *s e* . (*s*,*e*) ∈ *set pts*}
proof(*induction pts*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons p pts*) **thus** *?case* **by**(*cases p*, *simp*, *blast*)
qed

We can reuse the wordinterval theory to reason about ports

lemma *ports-to-set-wordinterval*: *ports-to-set ps* = *wordinterval-to-set* (*l2br ps*)
  by(*induction ps rule*: *l2br.induct*) (*auto*)

definition *ports-invert* :: *ipt-ports* ⇒ *ipt-ports* **where**
  ports-invert ps = *br2l* (*wordinterval-invert* (*l2br ps*))

lemma *ports-invert*: *ports-to-set* (*ports-invert ps*) = − *ports-to-set ps*
  by(*auto simp add*: *ports-invert-def l2br-br2l ports-to-set-wordinterval*)

end
theory *Simple-Packet*
imports *../Bitmagic/IPv4Addr Protocol*
begin

11 Simple Packet

Packet constants are prefixed with *p*

record *simple-packet* = *p-iiface* :: *string*
  p-oiface :: *string*
  p-src :: *ipv4addr*
  p-dst :: *ipv4addr*
  p-proto :: *primitive-protocol*
  p-sport :: *16 word*
  p-dport :: *16 word*

value (|*p-iiface* = *''eth1''*, *p-oiface* = *''''*, *p-src* = *0*, *p-dst* = *0*, *p-proto* = *TCP*, *p-sport* = *0*, *p-dport* = *0*|)
end
theory *Common-Primitive-Syntax*
imports *../Datatype-Selectors IpAddresses Iface Protocol Ports Simple-Packet*
begin

12 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.

datatype *common-primitive* =
  is-Src: *Src* (*src-sel*: *ipt-ipv4range*) |
  is-Dst: *Dst* (*dst-sel*: *ipt-ipv4range*) |
  is-Iiface: *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-ports*) |

is-Dst-Ports: *Dst-Ports* (*dst-ports-sel*: *ipt-ports*) |
is-Extra: *Extra* (*extra-sel*: *string*)

lemma *wf-disc-sel-common-primitive*[*simp*]:
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-Extra*, *extra-sel*) *Extra*
by(*simp-all add*: *wf-disc-sel.simps*)

— Example
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*|)

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

13 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*
unfolding *wf-unknown-match-tac-def* **by**(*simp add*: *fun-eq-iff*)

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*
unfolding *wf-unknown-match-tac-def* **by**(*simp add*: *fun-eq-iff*)

lemma *packet-independent-unknown-match-tacs*: *packet-independent-α in-doubt-allow*
packet-independent-α in-doubt-deny
by(*simp-all add*: *packet-independent-α-def*)

end
theory *Common-Primitive-Matcher*
imports *../Semantics-Ternary/Semantics-Ternary Common-Primitive-Syntax ../Bitmagic/IPv4Addr ../Semantics-Ternary/Unknown-Match-Tacs*
begin

13.1 Primitive Matchers: IP Port Iface Matcher

fun *common-matcher* :: (*common-primitive*, *simple-packet*) *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* ∈ *ipv4s-to-set ip*) |
common-matcher (*Dst ip*) *p* = *bool-to-ternary* (*p-dst p* ∈ *ipv4s-to-set ip*) |

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

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

common-matcher (*Extra -*) *p* = *TernaryUnknown*

Warning: beware of the sloppy term 'empty' portrange

An 'empty' port range means it can never match! Basically, *MatchNot* (*Match* (*Src-Ports* [(*0*, *0xFFFF*)])) is False

lemma ¬ *matches* (*common-matcher*, α) (*MatchNot* (*Match* (*Src-Ports* [(*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*|)

An 'empty' port range means it always matches! Basically, *MatchNot* (*Match* (*Src-Ports* [])) 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* []))) *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*|)

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

lemma *matches* (*common-matcher*, α) (*Match* (*Src-Ports* [(*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*|)
¬ *matches* (*common-matcher*, α) (*Match* (*Src-Ports* [(*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*|)
¬*matches* (*common-matcher*, α) (*MatchNot* (*Match* (*Src-Ports* [(*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*|)

Lemmas when matching on *Src* or *Dst*

lemma *common-matcher-SrcDst-defined*:
common-matcher (*Src m*) *p* ≠ *TernaryUnknown*
common-matcher (*Dst m*) *p* ≠ *TernaryUnknown*
common-matcher (*Src-Ports ps*) *p* ≠ *TernaryUnknown*
common-matcher (*Dst-Ports ps*) *p* ≠ *TernaryUnknown*
apply(*case-tac* [!] *m*)
apply(*simp-all add*: *bool-to-ternary-Unknown*)
done
lemma *common-matcher-SrcDst-defined-simp*:
common-matcher (*Src x*) *p* ≠ *TernaryFalse* ⟷ *common-matcher* (*Src x*) *p* = *TernaryTrue*
common-matcher (*Dst x*) *p* ≠ *TernaryFalse* ⟷ *common-matcher* (*Dst x*) *p* = *TernaryTrue*
apply (*metis eval-ternary-Not.cases common-matcher-SrcDst-defined*(*1*) *ternary-value.distinct*(*1*))
apply (*metis eval-ternary-Not.cases common-matcher-SrcDst-defined*(*2*) *ternary-value.distinct*(*1*))

done
lemma *match-simplematcher-SrcDst*:
matches (*common-matcher*, α) (*Match* (*Src X*)) *a p* $\longleftrightarrow$ *p-src p* $\in$ *ipv4s-to-set X*
matches (*common-matcher*, α) (*Match* (*Dst X*)) *a p* $\longleftrightarrow$ *p-dst p* $\in$ *ipv4s-to-set X*
apply(*simp-all add*: *matches-case-ternaryvalue-tuple split*: *ternaryvalue.split*)
apply (*metis bool-to-ternary.elims bool-to-ternary-Unknown ternaryvalue.distinct*(*1*))+
done
lemma *match-simplematcher-SrcDst-not*:
matches (*common-matcher*, α) (*MatchNot* (*Match* (*Src X*))) *a p* $\longleftrightarrow$ *p-src p* $\notin$ *ipv4s-to-set X*
matches (*common-matcher*, α) (*MatchNot* (*Match* (*Dst X*))) *a p* $\longleftrightarrow$ *p-dst p* $\notin$ *ipv4s-to-set X*
apply(*simp-all add*: *matches-case-ternaryvalue-tuple split*: *ternaryvalue.split*)
apply(*case-tac* [!] *X*)
apply(*simp-all add*: *bool-to-ternary-simps*)
done
lemma *common-matcher-SrcDst-Inter*:
($\forall$ *m*$\in$*set X*. *matches* (*common-matcher*, α) (*Match* (*Src m*)) *a p*) $\longleftrightarrow$ *p-src p* $\in$ ($\bigcap$ *x*$\in$*set X*. *ipv4s-to-set x*)
($\forall$ *m*$\in$*set X*. *matches* (*common-matcher*, α) (*Match* (*Dst m*)) *a p*) $\longleftrightarrow$ *p-dst p* $\in$ ($\bigcap$ *x*$\in$*set X*. *ipv4s-to-set x*)
by(*simp-all add*: *matches-case-ternaryvalue-tuple bool-to-ternary-Unknown bool-to-ternary-simps split*: *ternaryvalue.split*)
lemma *match-simplematcher-Iface*:
matches (*common-matcher*, α) (*Match* (*IIface X*)) *a p* $\longleftrightarrow$ *match-iface X* (*p-iiface p*)
matches (*common-matcher*, α) (*Match* (*OIface X*)) *a p* $\longleftrightarrow$ *match-iface X* (*p-oiface p*)
by(*simp-all add*: *matches-case-ternaryvalue-tuple bool-to-ternary-Unknown bool-to-ternary-simps split*: *ternaryvalue.split*)
lemma *match-simplematcher-Iface-not*:
matches (*common-matcher*, α) (*MatchNot* (*Match* (*IIface X*))) *a p* $\longleftrightarrow$ $\neg$ *match-iface X* (*p-iiface p*)
matches (*common-matcher*, α) (*MatchNot* (*Match* (*OIface X*))) *a p* $\longleftrightarrow$ $\neg$ *match-iface X* (*p-oiface p*)
by(*simp-all add*: *matches-case-ternaryvalue-tuple bool-to-ternary-simps split*: *ternaryvalue.split*)

multiport list is a way to express disjunction in one matchexpression in some firewalls

lemma *multiports-disjuction*:
($\exists$ *rg*$\in$*set spts*. *matches* (*common-matcher*, α) (*Match* (*Src-Ports* [*rg*])) *a p*) $\longleftrightarrow$
matches (*common-matcher*, α) (*Match* (*Src-Ports spts*)) *a p*
($\exists$ *rg*$\in$*set dpts*. *matches* (*common-matcher*, α) (*Match* (*Dst-Ports* [*rg*])) *a p*) $\longleftrightarrow$
matches (*common-matcher*, α) (*Match* (*Dst-Ports dpts*)) *a p*

apply(*simp-all add*: *bool-to-ternary-Unknown matches-case-ternaryvalue-tuple bunch-of-lemmata-about-matches bool-to-ternary-simps split*: *ternaryvalue.split ternaryvalue.split-asm*)
apply(*simp-all add*: *ports-to-set*)
apply(*safe*)
apply *force*+
done

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

lemma *common-matcher-MatchNot-Iface*:
matches (*common-matcher*, α) (*MatchNot* (*Match* (*IIface iface*))) *a p* $\longleftrightarrow$ $\neg$ *match-iface iface* (*p-iiface p*)
matches (*common-matcher*, α) (*MatchNot* (*Match* (*OIface iface*))) *a p* $\longleftrightarrow$ $\neg$ *match-iface iface* (*p-oiface p*)
by(*simp-all add*: *matches-case-ternaryvalue-tuple bool-to-ternary-simps split*: *ternaryvalue.split*)

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

fun *optimize-primitive-univ* :: *common-primitive match-expr* $\Rightarrow$ *common-primitive match-expr* **where**
optimize-primitive-univ (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) = *MatchAny* |
optimize-primitive-univ (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *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* (*Src-Ports* [(*s*, *e*)])) = (*if s* = *0* $\wedge$ *e* = *0xFFFF then MatchAny else* (*Match* (*Src-Ports* [(*s*, *e*)]))) |
optimize-primitive-univ (*Match* (*Dst-Ports* [(*s*, *e*)])) = (*if s* = *0* $\wedge$ *e* = *0xFFFF then MatchAny else* (*Match* (*Dst-Ports* [(*s*, *e*)]))) |
optimize-primitive-univ (*Match* (*Prot ProtoAny*)) = *MatchAny* |
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-correct-matchexpr*: *matches* (*common-matcher*, α) *m* = *matches* (*common-matcher*, α) (*optimize-primitive-univ m*)
apply(*simp add*: *fun-eq-iff*, *clarify*, *rename-tac a p*)
apply(*rule matches-iff-apply-f*)
apply(*simp*)

apply(*induction m rule*: *optimize-primitive-univ.induct*)
apply(*simp-all add*: *match-ifaceAny eval-ternary-simps ip-in-ipv4range-set-from-bitmask-UNIV eval-ternary-idempotence-Not bool-to-ternary-simps*)
apply(*subgoal-tac* (*max-word*::*16 word*) = *65535*,*simp*,*simp add*: *max-word-def*)+
done

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*
using *optimize-matches optimize-primitive-univ-correct-matchexpr* **by** *metis*

lemma *packet-independent-β-unknown-common-matcher*: *packet-independent-β-unknown common-matcher*
apply(*simp add*: *packet-independent-β-unknown-def*)
apply(*clarify*)
apply(*rename-tac A p1 p2*)
apply(*case-tac A*)
by(*simp-all add*: *bool-to-ternary-Unknown*)

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

fun *upper-closure-matchexpr* :: *action* ⇒ *common-primitive match-expr* ⇒ *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* ∨ *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* ∨ *a* = *Drop* ⟹ *remove-unknowns-generic* (*common-matcher*, *in-doubt-allow*) *a m* = *upper-closure-matchexpr a m*
by(*induction a m rule*: *upper-closure-matchexpr.induct*)
(*simp-all add*: *unknown-match-all-def unknown-not-match-any-def bool-to-ternary-Unknown*)

fun *lower-closure-matchexpr* :: *action* ⇒ *common-primitive match-expr* ⇒ *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 = *Accept* ∨ *a* = *Drop* ⟹ *remove-unknowns-generic* (*common-matcher*, *in-doubt-deny*) *a m* = *lower-closure-matchexpr a m*
by(*induction a m rule*: *lower-closure-matchexpr.induct*)
(*simp-all add*: *unknown-match-all-def unknown-not-match-any-def bool-to-ternary-Unknown*)

end
theory *Example-Semantics*
imports ../*Call-Return-Unfolding* ../*Primitive-Matchers*/*Common-Primitive-Matcher*
begin

14 Examples Big Step Semantics

we use a primitive matcher which always applies.

fun *applies-Yes* :: (*'a*, *'p*) *matcher* **where**
applies-Yes m p = *True*
lemma[*simp*]: *Semantics.matches applies-Yes MatchAny p* **by** *simp*
lemma[*simp*]: *Semantics.matches applies-Yes* (*Match e*) *p* **by** *simp*

definition *m*=*Match* (*Src* (*Ip4Addr* (*0*,*0*,*0*,*0*)))
lemma[*simp*]: *Semantics.matches applies-Yes m p* **by** (*simp add*: *m-def*)

lemma [*''FORWARD''* ↦ [(*Rule m Log*), (*Rule m Accept*), (*Rule m Drop*)]],*applies-Yes*,*p*⊢
⟨[*Rule MatchAny* (*Call ''FORWARD''*)], *Undecided*⟩ ⇒ (*Decision FinalAllow*)
apply(*rule call-result*)
apply(*auto*)
apply(*rule seq-cons*)
apply(*auto intro*:*Semantics.log*)
apply(*rule seq-cons*)
apply(*auto intro*: *Semantics.accept*)
apply(*rule Semantics.decision*)
done

lemma [*''FORWARD''* ↦ [(*Rule m Log*), (*Rule m* (*Call ''foo''*)), (*Rule m Accept*)],
''foo'' ↦ [(*Rule m Log*), (*Rule m Return*)]],*applies-Yes*,*p*⊢
⟨[*Rule MatchAny* (*Call ''FORWARD''*)], *Undecided*⟩ ⇒ (*Decision FinalAllow*)
apply(*rule call-result*)
apply(*auto*)
apply(*rule seq-cons*)
apply(*auto intro*: *Semantics.log*)
apply(*rule seq-cons*)
apply(*rule Semantics.call-return*[**where** rs_1=[*Rule m Log*] **and** rs_2=[]])
apply(*simp*)+
apply(*auto intro*: *Semantics.log*)
apply(*auto intro*: *Semantics.accept*)
done

lemma [*''FORWARD''* ↦ [*Rule m* (*Call ''foo''*), *Rule m Drop*], *''foo''* ↦ []],*applies-Yes*,*p*⊢
⟨[*Rule MatchAny* (*Call ''FORWARD''*)], *Undecided*⟩ ⇒ (*Decision FinalDeny*)
apply(*rule call-result*)
apply(*auto*)
apply(*rule Semantics.seq-cons*)
apply(*rule Semantics.call-result*)
apply(*auto*)
apply(*rule Semantics.skip*)
apply(*auto intro*: *deny*)
done

lemma $((\lambda rs.$ *process-call* $[''FORWARD'' \mapsto [Rule\ m\ (Call\ ''foo''), Rule\ m\ Drop],$
$''foo'' \mapsto []]\ rs)$ ^^$2)$
$[Rule\ MatchAny\ (Call\ ''FORWARD'')]$
$= [Rule\ (MatchAnd\ MatchAny\ m)\ Drop]$ **by** *eval*

hide-const *m*

definition $pkt=(\!|p\text{-}iiface=''+'', p\text{-}oiface=''+'', p\text{-}src=0, p\text{-}dst=0, p\text{-}proto=TCP,$
$p\text{-}sport=0,\ p\text{-}dport=0|\!)$

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* :: (*common-primitive*, *simple-packet*) *matcher*
where
applies-exampleMatchExact (*Src* (*Ip4Addr addr*)) $p \longleftrightarrow$ *p-src* $p =$ (*ipv4addr-of-dotdecimal addr*) |
applies-exampleMatchExact (*Dst* (*Ip4Addr addr*)) $p \longleftrightarrow$ *p-dst* $p =$ (*ipv4addr-of-dotdecimal addr*) |
applies-exampleMatchExact (*Prot ProtoAny*) $p \longleftrightarrow$ *True* |
applies-exampleMatchExact (*Prot* (*Proto TCP*)) $p \longleftrightarrow$ *p-proto* $p = TCP$ |
applies-exampleMatchExact (*Prot* (*Proto UDP*)) $p \longleftrightarrow$ *p-proto* $p = UDP$

lemma $[''FORWARD'' \mapsto [$ *Rule* (*MatchAnd* (*Match* (*Src* (*Ip4Addr* $(0,0,0,0)$))))
(*Match* (*Dst* (*Ip4Addr* $(0,0,0,0)$)))) *Reject*,
Rule (*Match* (*Dst* (*Ip4Addr* $(0,0,0,0)$))) *Log*,
Rule (*Match* (*Prot* (*Proto TCP*))) *Accept*,
Rule (*Match* (*Prot* (*Proto TCP*))) *Drop*]
],*applies-exampleMatchExact*, *pkt*(|*p-src*:=(*ipv4addr-of-dotdecimal* $(1,2,3,4)$),
p-dst:=(*ipv4addr-of-dotdecimal* $(0,0,0,0)$))|)$\vdash$
$\langle$[*Rule MatchAny* (*Call* $''FORWARD''$)], *Undecided*$\rangle \Rightarrow$ (*Decision FinalAllow*)
apply(*rule call-result*)
apply(*auto*)
apply(*rule Semantics.seq-cons*)
apply(*auto intro*: *Semantics.nomatch simp add*: *ipv4addr-of-dotdecimal.simps ipv4addr-of-nat-def*)
apply(*rule Semantics.seq-cons*)
apply(*auto intro*: *Semantics.log simp add*: *ipv4addr-of-dotdecimal.simps ipv4addr-of-nat-def*)
apply(*rule Semantics.seq-cons*)
apply(*auto simp add*: *pkt-def intro*: *Semantics.accept*)
apply(*auto intro*: *Semantics.decision*)
done

end
theory *Negation-Type-DNF*
imports *Negation-Type*

begin

15 Negation Type DNF

type-synonym *'a dnf* = *(('a negation-type) list) list*

fun *cnf-to-bool* :: *('a ⇒ bool) ⇒ 'a negation-type list ⇒ bool* **where**
cnf-to-bool - [] ⟷ True |
cnf-to-bool f (Pos a#as) ⟷ (f a) ∧ cnf-to-bool f as |
cnf-to-bool f (Neg a#as) ⟷ (¬ f a) ∧ cnf-to-bool f as

fun *dnf-to-bool* :: *('a ⇒ bool) ⇒ 'a dnf ⇒ bool* **where**
dnf-to-bool - [] ⟷ False |
dnf-to-bool f (as#ass) ⟷ (cnf-to-bool f as) ∨ (dnf-to-bool f ass)

representing *True*

definition *dnf-True* :: *'a dnf* **where**
dnf-True ≡ [[]]
lemma *dnf-True*: *dnf-to-bool f dnf-True*
unfolding *dnf-True-def* **by**(*simp*)

representing *False*

definition *dnf-False* :: *'a dnf* **where**
dnf-False ≡ []
lemma *dnf-False*: *¬ dnf-to-bool f dnf-False*
unfolding *dnf-False-def* **by**(*simp*)

lemma *cnf-to-bool-append*: *cnf-to-bool γ (a1 @ a2) ⟷ cnf-to-bool γ a1 ∧ cnf-to-bool γ a2*
by(*induction γ a1 rule*: *cnf-to-bool.induct*) (*simp-all*)
lemma *dnf-to-bool-append*: *dnf-to-bool γ (a1 @ a2) ⟷ dnf-to-bool γ a1 ∨ dnf-to-bool γ a2*
by(*induction a1*) (*simp-all*)

definition *dnf-and* :: *'a dnf ⇒ 'a dnf ⇒ '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 ⟷ (∀ c ∈ set cnf. (case c of Pos a ⇒ f a | Neg a ⇒ ¬ f a))*
proof(*induction cnf*)
case *Nil* **thus** *?case* **by** *simp*
next
case *Cons* **thus** *?case* **by** (*simp split*: *negation-type.split*)
qed
lemma *dnf-to-bool-set*: *dnf-to-bool γ dnf ⟷ (∃ d ∈ set dnf. cnf-to-bool γ d)*
proof(*induction dnf*)

case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons d d1*) **thus** *?case* **by**(*simp*)
qed

lemma *dnf-to-bool-seteq*: *set ‘ set d1 = set ‘ set d2* $\Longrightarrow$ *dnf-to-bool* γ *d1* $\longleftrightarrow$ *dnf-to-bool* γ *d2*
proof −
assume *assm*: *set ‘ set d1 = set ‘ set d2*
have *helper1*: $\bigwedge P\ d.\ (\exists d{\in}set\ d.\ \forall c{\in}set\ d.\ P\ c) \longleftrightarrow (\exists d{\in}set\text{ ‘ }set\ d.\ \forall c{\in}d.\ P\ c)$ **by** *blast*
from *assm* **show** *?thesis*
apply(*simp add*: *dnf-to-bool-set cnf-to-bool-set*)
apply(*subst helper1*)
apply(*subst helper1*)
apply(*simp*)
done
qed

lemma *dnf-and-correct*: *dnf-to-bool* γ (*dnf-and d1 d2*) $\longleftrightarrow$ *dnf-to-bool* γ *d1* $\wedge$ *dnf-to-bool* γ *d2*
apply(*simp add*: *dnf-and-def*)
apply(*induction d1*)
apply(*simp*)
apply(*simp add*: *dnf-to-bool-append*)
apply(*simp add*: *dnf-to-bool-set cnf-to-bool-set*)
by (*meson UnCI UnE*)

lemma *dnf-and-symmetric*: *dnf-to-bool* γ (*dnf-and d1 d2*) $\longleftrightarrow$ *dnf-to-bool* γ (*dnf-and d2 d1*)
using *dnf-and-correct* **by** *blast*

15.0.1 inverting a DNF

Example

lemma $(\neg\ ((a1 \wedge a2) \vee b \vee c)) = ((\neg a1 \wedge \neg\ b \wedge \neg\ c) \vee (\neg a2 \wedge \neg\ b \wedge \neg\ c))$ **by** *blast*
lemma $(\neg\ ((a1 \wedge a2) \vee (b1 \wedge b2) \vee c)) = ((\neg a1 \wedge \neg\ b1 \wedge \neg\ c) \vee (\neg a2 \wedge \neg\ b1 \wedge \neg\ c) \vee (\neg a1 \wedge \neg\ b2 \wedge \neg\ c) \vee (\neg a2 \wedge \neg\ b2 \wedge \neg\ c))$ **by** *blast*

fun *listprepend* :: *'a list* ⇒ *'a list list* ⇒ *'a list list* **where**
listprepend [] *ns* = [] |
listprepend (*a#as*) *ns* = (*map* (λ*xs*. *a#xs*) *ns*) @ (*listprepend as ns*)

lemma *listprepend* [*a*,*b*] [*as*, *bs*] = [*a#as*, *a#bs*, *b#as*, *b#bs*] **by** *simp*

lemma *map-a-and*: *dnf-to-bool* γ (*map* (*op* # *a*) *ds*) $\longleftrightarrow$ *dnf-to-bool* γ [[*a*]] $\wedge$ *dnf-to-bool* γ *ds*
apply(*induction ds*)

apply(*simp-all*)
apply(*case-tac a*)
apply(*simp-all*)
apply *blast*+
done

this is how *listprepend* works:

lemma $\neg$ *dnf-to-bool* γ (*listprepend* [] *ds*) **by**(*simp*)
lemma *dnf-to-bool* γ (*listprepend* [*a*] *ds*) $\longleftrightarrow$ *dnf-to-bool* γ [[*a*]] $\wedge$ *dnf-to-bool* γ *ds* **by**(*simp add*: *map-a-and*)
lemma *dnf-to-bool* γ (*listprepend* [*a*, *b*] *ds*) $\longleftrightarrow$ (*dnf-to-bool* γ [[*a*]] $\wedge$ *dnf-to-bool* γ *ds*) $\vee$ (*dnf-to-bool* γ [[*b*]] $\wedge$ *dnf-to-bool* γ *ds*)
by(*simp add*: *map-a-and dnf-to-bool-append*)

We use $\exists$ to model the big $\vee$ operation

lemma *listprepend-correct*: *dnf-to-bool* γ (*listprepend as ds*) $\longleftrightarrow$ ($\exists$ *a*$\in$ *set as*. *dnf-to-bool* γ [[*a*]] $\wedge$ *dnf-to-bool* γ *ds*)
apply(*induction as*)
apply(*simp*)
apply(*simp*)
apply(*rename-tac a as*)
apply(*simp add*: *map-a-and cnf-to-bool-append dnf-to-bool-append*)
by *blast*
lemma *listprepend-correct'*: *dnf-to-bool* γ (*listprepend as ds*) $\longleftrightarrow$ (*dnf-to-bool* γ (*map* (λ*a*. [*a*]) *as*) $\wedge$ *dnf-to-bool* γ *ds*)
apply(*induction as*)
apply(*simp*)
apply(*simp*)
apply(*rename-tac a as*)
apply(*simp add*: *map-a-and cnf-to-bool-append dnf-to-bool-append*)
by *blast*

lemma *cnf-invert-singelton*: *cnf-to-bool* γ [*invert a*] $\longleftrightarrow$ $\neg$ *cnf-to-bool* γ [*a*] **by**(*cases a*, *simp-all*)

lemma *cnf-singleton-false*: ($\exists$ *a'*$\in$*set as*. $\neg$ *cnf-to-bool* γ [*a'*]) $\longleftrightarrow$ $\neg$ *cnf-to-bool* γ *as*
by(*induction* γ *as rule*: *cnf-to-bool.induct*) (*simp-all*)

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*: *dnf-to-bool* γ (*dnf-not d*) $\longleftrightarrow$ $\neg$ *dnf-to-bool* γ *d*
apply(*induction d*)
apply(*simp-all*)
apply(*simp add*: *listprepend-correct*)
apply(*simp add*: *cnf-invert-singelton cnf-singleton-false*)
done

15.0.2 Optimizing

definition *optimize-dfn* :: *'a dnf* ⇒ *'a dnf* **where**
optimize-dfn dnf = *map remdups* (*remdups dnf*)

lemma *dnf-to-bool f* (*optimize-dfn dnf*) = *dnf-to-bool f dnf*
unfolding *optimize-dfn-def*
apply(*rule dnf-to-bool-seteq*)
apply(*simp*)
by (*metis image-cong image-image set-remdups*)

end
theory *Fixed-Action*
imports *Semantics-Ternary*
begin

16 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*:
$n > 0 \Longrightarrow$ *approximating-bigstep-fun* γ *p* (*r#rs*) *Undecided* = *approximating-bigstep-fun* γ *p* ((*replicate n r*)@*rs*) *Undecided*
apply(*induction n*)
apply(*simp*)
apply(*simp*)
apply(*case-tac r*)
apply(*rename-tac m a*)
apply(*simp split*: *action.split*)
by *fastforce*

utility lemmas

lemma *fixedaction-Log*: *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m Log*) *ms*) *Undecided* = *Undecided*
apply(*induction ms*, *simp-all*)
done
lemma *fixedaction-Empty*:*approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m Empty*) *ms*) *Undecided* = *Undecided*
apply(*induction ms*, *simp-all*)
done
lemma *helperX1-Log*: *matches* γ *m'* *Log p* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*map* ((λm. *Rule m Log*) ∘ *MatchAnd m'*) *m2'* @ *rs2*) *Undecided* =
approximating-bigstep-fun γ *p rs2 Undecided*
apply(*induction m2'*)
apply(*simp-all split*: *action.split*)

done
lemma *helperX1-Empty*: *matches* γ *m′ Empty p* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*map* ((λ*m*. *Rule m Empty*) $\circ$ *MatchAnd m′*) *m2′* @ *rs2*) *Undecided* =
approximating-bigstep-fun γ *p rs2 Undecided*
apply(*induction m2′*)
apply(*simp-all split*: *action.split*)
done
lemma *helperX3*: *matches* γ *m′ a p* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*map* ((λ*m*. *Rule m a*) $\circ$ *MatchAnd m′*) *m2′* @ *rs2*) *Undecided* =
approximating-bigstep-fun γ *p* (*map* (λ*m*. *Rule m a*) *m2′* @ *rs2*) *Undecided*
apply(*induction m2′*)
apply(*simp*)
apply(*case-tac a*)
apply(*simp-all add*: *matches-simps*)
done

lemmas *fixed-action-simps* = *helperX1-Log helperX1-Empty helperX3*
hide-fact *helperX1-Log helperX1-Empty helperX3*

lemma *fixedaction-swap*:
approximating-bigstep-fun γ *p* (*map* (λ*m*. *Rule m a*) (*m1*@*m2*)) *s* = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*m2*@*m1*)) *s*
proof(*cases s*)
case *Decision* **thus** *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*m1* @ *m2*)) *s* = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*m2* @ *m1*)) *s*
by(*simp add*: *Decision-approximating-bigstep-fun*)
next
case *Undecided*
have *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) *m1* @ *map* (λ*m*. *Rule m a*) *m2*) *Undecided* = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) *m2* @ *map* (λ*m*. *Rule m a*) *m1*) *Undecided*
proof(*induction m1*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons m m1*)
{ fix *m rs*
have *approximating-bigstep-fun* γ *p* ((*map* (λ*m*. *Rule m Log*) *m*)@*rs*) *Undecided* =
approximating-bigstep-fun γ *p rs Undecided*
by(*induction m*) (*simp-all*)
} note *Log-helper*=*this*
{ fix *m rs*
have *approximating-bigstep-fun* γ *p* ((*map* (λ*m*. *Rule m Empty*) *m*)@*rs*) *Undecided* =
approximating-bigstep-fun γ *p rs Undecided*
by(*induction m*) (*simp-all*)

} **note** *Empty-helper=this*

show *?case* (**is** *?goal*)
  proof(*cases matches* γ *m a p*)
    case *True*
      thus *?goal*
        proof(*induction m2*)
          case *Nil* **thus** *?case* **by** *simp*
        next
          case *Cons* **thus** *?case*
            apply(*simp split*:*action.split action.split-asm*)
            using *Log-helper Empty-helper* **by** *fastforce+*
        qed
    next
    case *False*
      thus *?goal*
       apply(*simp*)
       apply(*simp add*: *Cons.IH*)
       apply(*induction m2*)
        apply(*simp-all*)
       apply(*simp split*:*action.split action.split-asm*)
       apply *fastforce*
      **done*
  qed
qed
thus *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*m1* @ *m2*)) *s* = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*m2* @ *m1*)) *s* **using** *Undecided* **by** *simp*
qed

corollary *fixedaction-reorder*: *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*m1* @ *m2* @ *m3*)) *s* = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*m2* @ *m1* @ *m3*)) *s*
proof(*cases s*)
case *Decision* **thus** *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*m1* @ *m2* @ *m3*)) *s* = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*m2* @ *m1* @ *m3*)) *s*
  by(*simp add*: *Decision-approximating-bigstep-fun*)
next
case *Undecided*
have *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*m1* @ *m2* @ *m3*)) *Undecided* = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*m2* @ *m1* @ *m3*)) *Undecided*
  proof(*induction m3*)
    case *Nil* **thus** *?case* **using** *fixedaction-swap* **by** *fastforce*
    next
    case (*Cons m3'1 m3*)
      have *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) ((*m3'1* # *m3*) @ *m1* @ *m2*)) *Undecided* = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*)

((*m3'1* # *m3*) @ *m2* @ *m1*)) *Undecided*
apply(*simp*)
apply(*cases matches* γ *m3'1 a p*)
apply(*simp split*: *action.split action.split-asm*)
apply (*metis append-assoc fixedaction-swap map-append Cons.IH*)
apply(*simp*)
by (*metis append-assoc fixedaction-swap map-append Cons.IH*)
hence *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) ((*m1* @ *m2*) @ *m3'1* # *m3*)) *Undecided* = *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) ((*m2* @ *m1*) @ *m3'1* # *m3*)) *Undecided*
apply(*subst fixedaction-swap*)
apply(*subst*(*2*) *fixedaction-swap*)
by *simp*
thus *?case*
apply(*subst append-assoc*[*symmetric*])
apply(*subst append-assoc*[*symmetric*])
by *simp*
qed
thus *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) (*m1* @ *m2* @ *m3*)) *s* = *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) (*m2* @ *m1* @ *m3*)) *s*
using *Undecided* **by** *simp*
qed

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* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*map* (λ*m. Rule m a*) *m1*) *s* =
approximating-bigstep-fun γ *p* (*map* (λ*m. Rule m a*) *m2*) *s*
proof(*cases s*)
case *Decision* **thus** *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) *m1*) *s* = *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) *m2*) *s*
by(*simp add*: *Decision-approximating-bigstep-fun*)
next
case *Undecided*
assume *m1m2-seteq*: *set m1* = *set m2*
hence *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) *m1*) *Undecided* = *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) *m2*) *Undecided*
proof(*induction m1 arbitrary*: *m2*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons m m1*)
show *?case* (**is** *?goal*)
proof (*cases m* $\in$ *set m1*)
case *True*
from *True* **have** *set m1* = *set* (*m* # *m1*) **by** *auto*
from *Cons.IH*[*OF* ‹*set m1* = *set* (*m* # *m1*)›] **have** *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) (*m* # *m1*)) *Undecided* = *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) (*m1*)) *Undecided* **..**
thus *?goal* **by** (*metis Cons.IH Cons.prems* ‹*set m1* = *set* (*m* # *m1*)›)

next
case *False*
from *False* **have** $m \notin$ *set m1* **.**
show *?goal*
proof (*cases* $m \notin$ *set m2*)
case *True*
from *True* ‹$m \notin$ *set m1*› *Cons.prems* **have** *set m1* = *set m2* **by** *auto*
from *Cons.IH*[*OF this*] **show** *?goal* **by** (*metis Cons.IH Cons.prems* ‹*set m1* = *set m2*›)
next
case *False*
hence $m \in$ *set m2* **by** *simp*

have *repl-filter-simp*: (*replicate* (*length* [$x \leftarrow$*m2* . $x = m$]) *m*) = [$x \leftarrow$*m2* . $x = m$]
by (*metis* (*lifting*, *full-types*) *filter-set member-filter replicate-length-same*)

from *Cons.prems* ‹$m \notin$ *set m1*› **have** *set m1* = *set* (*filter* ($\lambda x.\ x \neq m$) *m2*) **by** *auto*
from *Cons.IH*[*OF this*] **have** *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) *m1*) *Undecided* = *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) [$x \leftarrow$*m2* . $x \neq m$]) *Undecided* **.**
from *this* **have** *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) (*m*#*m1*)) *Undecided* = *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) (*m*#[$x \leftarrow$*m2* . $x \neq m$])) *Undecided*
apply(*simp split*: *action.split*)
by *fast*
also have ... = *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) ([$x \leftarrow$*m2* . $x = m$]@[$x \leftarrow$*m2* . $x \neq m$])) *Undecided*
apply(*simp only*: *list.map*)
thm *approximating-bigstep-fun-prepend-replicate*[**where** *n*=*length* [$x \leftarrow$*m2* . $x = m$]]
apply(*subst approximating-bigstep-fun-prepend-replicate*[**where** *n*=*length* [$x \leftarrow$*m2* . $x = m$]])
apply (*metis* (*full-types*) *False filter-empty-conv neq0-conv repl-filter-simp replicate-0*)
by (*metis* (*lifting*, *no-types*) *map-append map-replicate repl-filter-simp*)
also have ... = *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) *m2*) *Undecided*
proof(*induction m2*)
case *Nil* **thus** *?case* **by** *simp*
next
case(*Cons m2'1 m2'*)
have *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) [$x \leftarrow$*m2'* . $x = m$] @ *Rule m2'1 a* # *map* (λm. *Rule m a*) [$x \leftarrow$*m2'* . $x \neq m$]) *Undecided* =
approximating-bigstep-fun γ *p* (*map* (λm. *Rule m a*) ([$x \leftarrow$*m2'* . $x = m$] @ [*m2'1*] @ [$x \leftarrow$*m2'* . $x \neq m$])) *Undecided* **by** *fastforce*
also have ... = *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) ([*m2'1*] @ [$x \leftarrow$*m2'* . $x = m$] @ [$x \leftarrow$*m2'* . $x \neq m$])) *Undecided*

using *fixedaction-reorder* **by** *fast*
finally have *XX*: *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) [*x*←*m2' . x = m*] @ *Rule m2'1 a # map* (λ*m. Rule m a*) [*x*←*m2' . x* $\neq$ *m*]) *Undecided* =
approximating-bigstep-fun γ *p* (*Rule m2'1 a #* (*map* (λ*m. Rule m a*) [*x*←*m2' . x = m*] @ *map* (λ*m. Rule m a*) [*x*←*m2' . x* $\neq$ *m*])) *Undecided*
by *fastforce*
from *Cons* **show** *?case*
apply(*case-tac m2'1 = m*)
apply(*simp split*: *action.split*)
apply *fast*
apply(*simp del*: *approximating-bigstep-fun.simps*)
apply(*simp only*: *XX*)
apply(*case-tac matches* γ *m2'1 a p*)
apply(*simp*)
apply(*simp split*: *action.split*)
apply(*fast*)
apply(*simp*)
done
qed
finally show *?goal* **.**
qed
qed
qed
thus *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) *m1*) *s* = *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) *m2*) *s* **using** *Undecided m1m2-seteq* **by** *simp*
qed

16.1 *match-list*

Reducing the firewall semantics to short-circuit matching evaluation

fun *match-list* :: (*'a*, *'packet*) *match-tac* ⇒ *'a match-expr list* ⇒ *action* ⇒ *'packet* ⇒ *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-list* γ *ms a p* ⟷ ($\exists$ *m* $\in$ *set ms. matches* γ *m a p*)
by(*induction ms*, *simp-all*)

lemma *match-list-True*: *match-list* γ *ms a p* ⟹ *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) *ms*) *Undecided* = (*case a of Accept* ⇒ *Decision FinalAllow*
| *Drop* ⇒ *Decision FinalDeny*
| *Reject* ⇒ *Decision FinalDeny*
| *Log* ⇒ *Undecided*
| *Empty* ⇒ *Undecided*
(∗*unhandled cases*∗)

)
apply(*induction ms*)
apply(*simp*)
apply(*simp split*: *split-if-asm action.split*)
apply(*simp add*: *fixedaction-Log fixedaction-Empty*)
done

lemma *match-list-False*: $\neg$ *match-list* γ *ms a p* $\Longrightarrow$ *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) *ms*) *Undecided* = *Undecided*
apply(*induction ms*)
apply(*simp*)
apply(*simp split*: *split-if-asm action.split*)
done

The key idea behind *match-list*: Reducing semantics to match list

lemma *match-list-semantics*: *match-list* γ *ms1 a p* $\longleftrightarrow$ *match-list* γ *ms2 a p* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*map* (λ*m. Rule m a*) *ms1*) *s* = *approximating-bigstep-fun* γ *p* (*map* (λ*m. Rule m a*) *ms2*) *s*
apply(*case-tac s*)
prefer *2*
apply(*simp add*: *Decision-approximating-bigstep-fun*)
apply(*simp*)
apply(*thin-tac s* = *Undecided*)
apply(*induction ms2*)
apply(*simp*)
apply(*induction ms1*)
apply(*simp*)
apply(*simp split*: *split-if-asm*)
apply(*rename-tac m ms2*)
apply(*simp del*: *approximating-bigstep-fun.simps*)
apply(*simp split*: *split-if-asm del*: *approximating-bigstep-fun.simps*)
apply(*simp split*: *action.split add*: *match-list-True fixedaction-Log fixedaction-Empty*)
apply(*simp*)
done

We can exploit de-morgan to get a disjunction in the match expression!

fun *match-list-to-match-expr* :: *'a match-expr list* $\Rightarrow$ *'a match-expr* **where**
match-list-to-match-expr [] = *MatchNot MatchAny* |
match-list-to-match-expr (*m#ms*) = *MatchNot* (*MatchAnd* (*MatchNot m*) (*MatchNot* (*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-list* γ *ms a p* $\longleftrightarrow$ *matches* γ (*match-list-to-match-expr ms*) *a p*

apply(*induction ms rule*: *match-list-to-match-expr.induct*)
apply(*simp add*: *bunch-of-lemmata-about-matches*)
apply(*simp*)
apply (*metis matches-DeMorgan matches-not-idem*)+
done

lemma *match-list-singleton*: *match-list* γ [*m*] *a p* $\longleftrightarrow$ *matches* γ *m a p* **by**(*simp*)

lemma *empty-concat*: (*concat* (*map* (λx. []) *ms*)) = []
apply(*induction ms*)
by(*simp-all*)

lemma *match-list-append*: *match-list* γ (*m1*@*m2*) *a p* $\longleftrightarrow$ ($\neg$ *match-list* γ *m1* *a p* $\longrightarrow$ *match-list* γ *m2 a p*)
apply(*induction m1*)
apply(*simp*)
apply(*simp*)
done

lemma *match-list-helper1*: $\neg$ *matches* γ *m2 a p* $\Longrightarrow$ *match-list* γ (*map* (λx. *MatchAnd x m2*) *m1'*) *a p* $\Longrightarrow$ *False*
apply(*induction m1'*)
apply(*simp*)
apply(*simp split*:*split-if-asm*)
by(*auto dest*: *matches-dest*)
lemma *match-list-helper2*: $\neg$ *matches* γ *m a p* $\Longrightarrow$ $\neg$ *match-list* γ (*map* (*MatchAnd* *m*) *m2'*) *a p*
apply(*induction m2'*)
apply(*simp*)
apply(*simp split*:*split-if-asm*)
by(*auto dest*: *matches-dest*)
lemma *match-list-helper3*: *matches* γ *m a p* $\Longrightarrow$ *match-list* γ *m2'* *a p* $\Longrightarrow$ *match-list* γ (*map* (*MatchAnd m*) *m2'*) *a p*
apply(*induction m2'*)
apply(*simp*)
apply(*simp split*:*split-if-asm*)
by (*simp add*: *matches-simps*)
lemma *match-list-helper4*: $\neg$ *match-list* γ *m2'* *a p* $\Longrightarrow$ $\neg$ *match-list* γ (*map* (*MatchAnd aa*) *m2'*) *a p*
apply(*induction m2'*)
apply(*simp*)
apply(*simp split*:*split-if-asm*)
by(*auto dest*: *matches-dest*)
lemma *match-list-helper5*: $\neg$ *match-list* γ *m2'* *a p* $\Longrightarrow$ $\neg$ *match-list* γ (*concat* (*map* (λx. *map* (*MatchAnd x*) *m2'*) *m1'*)) *a p*
apply(*induction m2'*)
apply(*simp add*:*empty-concat*)
apply(*simp split*:*split-if-asm*)
apply(*induction m1'*)

apply(*simp*)
apply(*simp add*: *match-list-append*)
by(*auto dest*: *matches-dest*)
lemma *match-list-helper6*: $\neg$ *match-list* γ *m1'* *a p* $\Longrightarrow$ $\neg$ *match-list* γ (*concat* (*map* (λx. *map* (*MatchAnd x*) *m2'*) *m1'*)) *a p*
apply(*induction m2'*)
apply(*simp add*:*empty-concat*)
apply(*simp split*:*split-if-asm*)
apply(*induction m1'*)
apply(*simp*)
apply(*simp add*: *match-list-append split*: *split-if-asm*)
by(*auto dest*: *matches-dest*)

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*: *matches* γ *m1 a p* = *match-list* γ *m1' a p* $\Longrightarrow$
matches γ (*MatchAnd m1 m2*) *a p* $\longleftrightarrow$ *match-list* γ (*map* (λx. *MatchAnd x m2*) *m1'*) *a p*
apply(*induction m1'*)
apply(*auto dest*: *matches-dest*)[*1*]
apply(*simp split*: *split-if-asm*)
apply *safe*
apply(*simp-all add*: *matches-simps*)
apply(*auto dest*: *match-list-helper*(*1*))[*1*]
by(*auto dest*: *matches-dest*)

lemma *matches-list-And-concat*: *matches* γ *m1 a p* = *match-list* γ *m1' a p* $\Longrightarrow$ *matches* γ *m2 a p* = *match-list* γ *m2' a p* $\Longrightarrow$
matches γ (*MatchAnd m1 m2*) *a p* $\longleftrightarrow$ *match-list* γ [*MatchAnd x y*. *x* <− *m1'*, *y* <− *m2'*] *a p*
apply(*induction m1'*)
apply(*auto dest*: *matches-dest*)[*1*]
apply(*simp split*: *split-if-asm*)
prefer *2*
apply(*simp add*: *match-list-append*)
apply(*subgoal-tac* $\neg$ *match-list* γ (*map* (*MatchAnd aa*) *m2'*) *a p*)
apply(*simp*)
apply *safe*
apply(*simp-all add*: *matches-simps match-list-append match-list-helper*)
done

lemma *fixedaction-wf-ruleset*: *wf-ruleset* γ *p* (*map* (λm. *Rule m a*) *ms*) $\longleftrightarrow$ $\neg$ *match-list* γ *ms a p* $\vee$ $\neg$ ($\exists$ *chain*. *a* = *Call chain*) $\wedge$ *a* $\neq$ *Return* $\wedge$ *a* $\neq$ *Unknown*
proof −
have *helper*: $\bigwedge a\ b\ c.\ a \longleftrightarrow c \Longrightarrow (a \longrightarrow b) = (c \longrightarrow b)$ **by** *fast*

show *?thesis*
apply(*simp add*: *wf-ruleset-def*)
apply(*rule helper*)
apply(*induction ms*)
apply(*simp*)
apply(*simp*)
done
qed

lemma *wf-ruleset-singleton*: *wf-ruleset* γ *p* [*Rule m a*] $\longleftrightarrow$ $\neg$ *matches* γ *m a p* $\vee$ $\neg$ ($\exists$ *chain*. *a* = *Call chain*) $\wedge$ *a* $\neq$ *Return* $\wedge$ *a* $\neq$ *Unknown*
by(*simp add*: *wf-ruleset-def*)

17 Normalized (DNF) matches

simplify a match expression. The output is a list of match exprissions, 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 *match-list-normalize-match*: *match-list* γ [*m*] *a p* $\longleftrightarrow$ *match-list* γ (*normalize-match m*) *a p*
proof(*induction m rule*:*normalize-match.induct*)
case *1* **thus** *?case* **by**(*simp add*: *match-list-singleton*)
next
case *2* **thus** *?case* **by**(*simp add*: *match-list-singleton*)
next
case (*3 m1 m2*) **thus** *?case*
apply(*simp-all add*: *match-list-singleton del*: *match-list.simps*(*2*))
apply(*case-tac matches* γ *m1 a p*)
apply(*rule matches-list-And-concat*)
apply(*simp*)
apply(*case-tac* (*normalize-match m1*))
apply *simp*
apply (*auto*)[*1*]
apply(*simp add*: *bunch-of-lemmata-about-matches match-list-helper*)
done
next
case *4* **thus** *?case*
apply(*simp-all add*: *match-list-singleton del*: *match-list.simps*(*2*))

apply(*simp add*: *match-list-append*)
apply(*safe*)
apply(*simp-all add*: *matches-DeMorgan*)
done
next
case *5* **thus** *?case*
apply(*simp-all add*: *match-list-singleton del*: *match-list.simps*(*2*))
apply (*metis matches-not-idem*)
done
next
case *6* **thus** *?case*
apply(*simp-all add*: *match-list-singleton del*: *match-list.simps*(*2*))
by (*metis bunch-of-lemmata-about-matches*(*3*))
next
case *7* **thus** *?case* **by**(*simp add*: *match-list-singleton*)
qed

thm *match-list-normalize-match*[*simplified match-list-singleton*]

theorem *normalize-match-correct*: *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*normalize-match m*)) *s* = *approximating-bigstep-fun* γ *p* [*Rule m a*] *s*
apply(*rule match-list-semantics*[*of - - - -* [*m*], *simplified*])
using *match-list-normalize-match* **by** *fastforce*

lemma *normalize-match-empty*: *normalize-match m* = [] $\Longrightarrow$ $\neg$ *matches* γ *m a p*
proof(*induction m rule*: *normalize-match.induct*)
case *3* **thus** *?case* **by**(*fastforce dest*: *matches-dest*)
next
case *4* **thus** *?case* **using** *match-list-normalize-match* **by** (*simp add*: *matches-DeMorgan*)
next
case *5* **thus** *?case* **using** *matches-not-idem* **by** *fastforce*
next
case *6* **thus** *?case* **by**(*simp add*: *bunch-of-lemmata-about-matches*)
qed(*simp-all*)

lemma *matches-to-match-list-normalize*: *matches* γ *m a p* = *match-list* γ (*normalize-match m*) *a p*
using *match-list-normalize-match*[*simplified match-list-singleton*] **.**

lemma *wf-ruleset-normalize-match*: *wf-ruleset* γ *p* [(*Rule m a*)] $\Longrightarrow$ *wf-ruleset* γ *p* (*map* (λ*m*. *Rule m a*) (*normalize-match m*))
proof(*induction m rule*: *normalize-match.induct*)
case *1* **thus** *?case* **by** *simp*
next
case *2* **thus** *?case* **by** *simp*
next

case *3* **thus** *?case* **by**(*simp add*: *fixedaction-wf-ruleset wf-ruleset-singleton matches-to-match-list-normalize*)
next
case *4* **thus** *?case*
apply(*simp add*: *wf-ruleset-append*)
apply(*simp add*: *fixedaction-wf-ruleset*)
apply(*unfold wf-ruleset-singleton*)
apply(*safe*)
apply(*simp-all add*: *matches-to-match-list-normalize*)
apply(*simp-all add*: *match-list-append*)
done
next
case *5* **thus** *?case* **by**(*simp add*: *wf-ruleset-singleton matches-to-match-list-normalize*)
next
case *6* **thus** *?case* **by**(*simp add*: *wf-ruleset-def*)
next
case *7* **thus** *?case* **by**(*simp-all add*: *wf-ruleset-append*)
qed

lemma *normalize-match-wf-ruleset*: *wf-ruleset* γ *p* (*map* (λ*m*. *Rule m a*) (*normalize-match m*)) $\Longrightarrow$ *wf-ruleset* γ *p* [*Rule m a*]
proof(*induction m rule*: *normalize-match.induct*)
case *1* **thus** *?case* **by** *simp*
next
case *2* **thus** *?case* **by** *simp*
next
case *3* **thus** *?case* **by**(*simp add*: *fixedaction-wf-ruleset wf-ruleset-singleton matches-to-match-list-normalize*)
next
case *4* **thus** *?case*
apply(*simp add*: *wf-ruleset-append*)
apply(*simp add*: *fixedaction-wf-ruleset*)
apply(*unfold wf-ruleset-singleton*)
apply(*safe*)
apply(*simp-all add*: *matches-to-match-list-normalize*)
apply(*simp-all add*: *match-list-append*)
done
next
case *5* **thus** *?case*
unfolding *wf-ruleset-singleton* **by**(*simp add*: *matches-to-match-list-normalize*)
next
case *6* **thus** *?case* **unfolding** *wf-ruleset-singleton* **using** *bunch-of-lemmata-about-matches*(*3*) **by** *metis*
next
case *7* **thus** *?case* **by**(*simp-all add*: *wf-ruleset-append*)
qed

lemma *good-ruleset-normalize-match*: *good-ruleset* [(*Rule m a*)] $\Longrightarrow$ *good-ruleset* (*map* (λ*m*. *Rule m a*) (*normalize-match m*))
by(*simp add*: *good-ruleset-def*)

18 Normalizing rules instead of only match expressions

fun *normalize-rules* :: (*'a match-expr* ⇒ *'a match-expr list*) ⇒ *'a rule list* ⇒ *'a rule list* **where**
normalize-rules - [] = [] |
normalize-rules f ((*Rule m a*)#*rs*) = (*map* (*λm. Rule m a*) (*f m*))@(*normalize-rules f rs*)

lemma *normalize-rules-singleton*: *normalize-rules f* [*Rule m a*] = *map* (*λm. Rule m a*) (*f m*) **by**(*simp*)
lemma *normalize-rules-fst*: (*normalize-rules f* (*r* # *rs*)) = (*normalize-rules f* [*r*]) @ (*normalize-rules f rs*)
by(*cases r*) (*simp*)

lemma *good-ruleset-normalize-rules*: *good-ruleset rs* ⟹ *good-ruleset* (*normalize-rules f rs*)
proof(*induction rs*)
case *Nil* **thus** *?case* **by** (*simp*)
next
case(*Cons r rs*)
from *Cons* **have** *IH*: *good-ruleset* (*normalize-rules f rs*) **using** *good-ruleset-tail* **by** *blast*
from *Cons.prems* **have** *good-ruleset* [*r*] **using** *good-ruleset-fst* **by** *fast*
hence *good-ruleset* (*normalize-rules f* [*r*]) **by**(*cases r*) (*simp add*: *good-ruleset-alt*)
with *IH good-ruleset-append* **have** *good-ruleset* (*normalize-rules f* [*r*] @ *normalize-rules f rs*) **by** *blast*
thus *?case* **using** *normalize-rules-fst* **by** *metis*
qed

lemma *simple-ruleset-normalize-rules*: *simple-ruleset rs* ⟹ *simple-ruleset* (*normalize-rules f rs*)
proof(*induction rs*)
case *Nil* **thus** *?case* **by** (*simp*)
next
case(*Cons r rs*)
from *Cons* **have** *IH*: *simple-ruleset* (*normalize-rules f rs*) **using** *simple-ruleset-tail* **by** *blast*
from *Cons.prems* **have** *simple-ruleset* [*r*] **using** *simple-ruleset-append* **by** *fastforce*
hence *simple-ruleset* (*normalize-rules f* [*r*]) **by**(*cases r*) (*simp add*: *simple-ruleset-def*)

with *IH simple-ruleset-append* **have** *simple-ruleset* (*normalize-rules f* [*r*] @ *normalize-rules f rs*) **by** *blast*
thus *?case* **using** *normalize-rules-fst* **by** *metis*
qed

lemma *normalize-rules-match-list-semantics*:
assumes $\forall m\ a.$ *match-list* γ $(f\ m)$ a p $=$ *matches* γ m a p **and** *simple-ruleset* *rs*
shows *approximating-bigstep-fun* γ p (*normalize-rules* f *rs*) s $=$ *approximating-bigstep-fun* γ p *rs* s
proof −
{ fix m a s
from *assms*(*1*) **have** *match-list* γ $(f\ m)$ a p $\longleftrightarrow$ *match-list* γ $[m]$ a p **by** *simp*
with *match-list-semantics*[*of* γ f m a p $[m]$] **have**
approximating-bigstep-fun γ p (*map* ($\lambda m.$ *Rule* m a) $(f\ m)$) s $=$ *approximating-bigstep-fun* γ p [*Rule* m a] s **by** *simp*
} note *ar*=*this* **{**
fix r s
from *ar*[*of get-action* r *get-match* r] **have**
(*approximating-bigstep-fun* γ p (*normalize-rules* f $[r]$) s) $=$ *approximating-bigstep-fun* γ p $[r]$ s
by(*cases* r) (*simp*)
} note *a*=*this*

note *a*=*this*

from *assms*(*2*) **show** *?thesis*
proof(*induction rs arbitrary*: s)
case *Nil* **thus** *?case* **by** (*simp*)
next
case (*Cons* r *rs*)
from *Cons.prems* **have** *simple-ruleset* $[r]$ **by**(*simp add*: *simple-ruleset-def*)
with *simple-imp-good-ruleset good-imp-wf-ruleset* **have** *wf-r*: *wf-ruleset* γ p $[r]$ **by** *fast*

from ‹*simple-ruleset* $[r]$› *simple-imp-good-ruleset good-imp-wf-ruleset* **have** *wf-r*:
wf-ruleset γ p $[r]$ **by** *fast*
from *simple-ruleset-normalize-rules*[*OF* ‹*simple-ruleset* $[r]$›] **have** *simple-ruleset* (*normalize-rules* f $[r]$)
by(*simp*)
with *simple-imp-good-ruleset good-imp-wf-ruleset* **have** *wf-nr*: *wf-ruleset* γ p (*normalize-rules* f $[r]$) **by** *fast*

from *Cons* **have** *IH*: $\bigwedge s.$ *approximating-bigstep-fun* γ p (*normalize-rules* f *rs*) s $=$ *approximating-bigstep-fun* γ p *rs* s
using *simple-ruleset-tail* **by** *force*

show *?case*
apply(*subst normalize-rules-fst*)
apply(*simp add*: *approximating-bigstep-fun-seq-wf*[*OF wf-nr*])
apply(*subst approximating-bigstep-fun-seq-wf*[*OF wf-r*, *simplified*])
apply(*simp add*: *a*)

apply(*simp add*: *IH*)
done
qed
qed

lemma *normalize-rules-match-list-semantics-2*:
assumes $\forall$ *r* $\in$ *set rs*. *match-list* γ (*f* (*get-match r*)) (*get-action r*) *p* = *matches* γ (*get-match r*) (*get-action r*) *p* **and** *simple-ruleset rs*
shows *approximating-bigstep-fun* γ *p* (*normalize-rules f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof −
{ fix *r s*
assume *r* $\in$ *set rs*
with *assms*(*1*) **have** *match-list* γ (*f* (*get-match r*)) (*get-action r*) *p* $\longleftrightarrow$ *match-list* γ [(*get-match r*)] (*get-action r*) *p* **by** *simp*
with *match-list-semantics*[*of* γ *f* (*get-match r*) (*get-action r*) *p* [(*get-match r*)]] **have**
approximating-bigstep-fun γ *p* (*map* (λ*m*. *Rule m* (*get-action r*)) (*f* (*get-match r*))) *s* =
approximating-bigstep-fun γ *p* [*Rule* (*get-match r*) (*get-action r*)] *s* **by** *simp*
hence (*approximating-bigstep-fun* γ *p* (*normalize-rules f* [*r*]) *s*) = *approximating-bigstep-fun* γ *p* [*r*] *s*
by(*cases r*) (*simp*)
}

with *assms* **show** *?thesis*
proof(*induction rs arbitrary*: *s*)
case *Nil* **thus** *?case* **by** (*simp*)
next
case (*Cons r rs*)
from *Cons.prems* **have** *simple-ruleset* [*r*] **by**(*simp add*: *simple-ruleset-def*)
with *simple-imp-good-ruleset good-imp-wf-ruleset* **have** *wf-r*: *wf-ruleset* γ *p* [*r*] **by** *fast*

from ‹*simple-ruleset* [*r*]› *simple-imp-good-ruleset good-imp-wf-ruleset* **have** *wf-r*:
wf-ruleset γ *p* [*r*] **by** *fast*
from *simple-ruleset-normalize-rules*[*OF* ‹*simple-ruleset* [*r*]›] **have** *simple-ruleset* (*normalize-rules f* [*r*])
by(*simp*)
with *simple-imp-good-ruleset good-imp-wf-ruleset* **have** *wf-nr*: *wf-ruleset* γ *p* (*normalize-rules f* [*r*]) **by** *fast*

from *Cons* **have** *IH*: $\bigwedge$*s*. *approximating-bigstep-fun* γ *p* (*normalize-rules f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
using *simple-ruleset-tail* **by** *force*

from *Cons* **have** *a*: $\bigwedge$*s*. *approximating-bigstep-fun* γ *p* (*normalize-rules f*

[r]) s = *approximating-bigstep-fun* γ p [r] s **by** *simp*

show *?case*
apply(*subst normalize-rules-fst*)
apply(*simp add*: *approximating-bigstep-fun-seq-wf*[*OF wf-nr*])
apply(*subst approximating-bigstep-fun-seq-wf*[*OF wf-r*, *simplified*])
apply(*simp add*: *a*)
apply(*simp add*: *IH*)
done
qed
qed

applying a function (with a prerequisite Q) to all rules

lemma *normalize-rules-property*:
assumes ∀ m ∈ *get-match* ‘ *set rs*. P m
and ∀ m. P m ⟶ (∀ m′ ∈ *set* (f m). Q m′)
shows ∀ m ∈ *get-match* ‘ *set* (*normalize-rules f rs*). Q m
proof
fix m **assume** a: m ∈ *get-match* ‘ *set* (*normalize-rules f rs*)
from *a assms* **show** Q m
proof(*induction rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
{
assume m ∈ *get-match* ‘ *set* (*normalize-rules f rs*)
from *Cons.IH this* **have** Q m **using** *Cons.prems*(*2*) *Cons.prems*(*3*) **by** *fastforce*
} **note** *1*=*this*
{
assume m ∈ *get-match* ‘ *set* (*normalize-rules f* [r])
hence a: m ∈ *set* (f (*get-match* r)) **by**(*cases* r) (*auto*)
with *Cons.prems*(*2*) *Cons.prems*(*3*) **have** ∀ m′∈*set* (f (*get-match* r)). Q m′ **by** *auto*
with a **have** Q m **by** *blast*
} **note** *2*=*this*
from *Cons.prems*(*1*) **have** m ∈ *get-match* ‘ *set* (*normalize-rules f* [r]) ∨ m ∈ *get-match* ‘ *set* (*normalize-rules f rs*)
by(*subst*(*asm*) *normalize-rules-fst*) *auto*
with *1 2* **show** *?case*
by(*elim disjE*)(*simp*)
qed
qed

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** ∀ m ∈ *get-match* ‘ *set rs*. P m
and ∀ m. P m ⟶ (∀ m′ ∈ *set* (f m). P m′)
shows ∀ m ∈ *get-match* ‘ *set* (*normalize-rules f rs*). P m

using *normalize-rules-property*[*OF assms*(*1*) *assms*(*2*)] **.**

lemma *normalize-rules-preserves'*: $\forall\ m \in$ *set rs*. P (*get-match m*) $\Longrightarrow \forall\, m.\ P\ m \longrightarrow (\forall\, m' \in$ *set* $(f\ m).\ P\ m') \Longrightarrow \forall\, m \in$ *set* (*normalize-rules f rs*). P (*get-match m*)
using *normalize-rules-preserves*[*simplified*] **by** *blast*

fun *normalize-rules-dnf* :: *'a rule list* $\Rightarrow$ *'a rule list* **where**
normalize-rules-dnf [] = [] |
normalize-rules-dnf ((*Rule m a*)#*rs*) = (*map* (λm. *Rule m a*) (*normalize-match m*))@(*normalize-rules-dnf rs*)

lemma *normalize-rules-dnf-def2*: *normalize-rules-dnf* = *normalize-rules normalize-match*
proof(*simp add*: *fun-eq-iff*, *intro allI*)
fix *x*::*'a rule list* **show** *normalize-rules-dnf x* = *normalize-rules normalize-match x*
proof(*induction x*)
case (*Cons r rs*) **thus** *?case* **by** (*cases r*) *simp*
qed(*simp*)
qed

lemma *wf-ruleset-normalize-rules-dnf*: *wf-ruleset* γ *p rs* $\Longrightarrow$ *wf-ruleset* γ *p* (*normalize-rules-dnf rs*)
proof(*induction rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case(*Cons r rs*)
from *Cons* **have** *IH*: *wf-ruleset* γ *p* (*normalize-rules-dnf rs*) **by**(*auto dest*: *wf-rulesetD*)
from *Cons.prems* **have** *wf-ruleset* γ *p* [*r*] **by**(*auto dest*: *wf-rulesetD*)
hence *wf-ruleset* γ *p* (*normalize-rules-dnf* [*r*]) **using** *wf-ruleset-normalize-match* **by**(*cases r*) *simp*
with *IH wf-ruleset-append* **have** *wf-ruleset* γ *p* (*normalize-rules-dnf* [*r*] @ *normalize-rules-dnf rs*) **by** *fast*
thus *?case* **using** *normalize-rules-dnf-def2 normalize-rules-fst* **by** *metis*
qed

lemma *good-ruleset-normalize-rules-dnf*: *good-ruleset rs* $\Longrightarrow$ *good-ruleset* (*normalize-rules-dnf rs*)
using *normalize-rules-dnf-def2 good-ruleset-normalize-rules* **by** *metis*

lemma *simple-ruleset-normalize-rules-dnf*: *simple-ruleset rs* $\Longrightarrow$ *simple-ruleset* (*normalize-rules-dnf rs*)
using *normalize-rules-dnf-def2 simple-ruleset-normalize-rules* **by** *metis*

lemma *simple-ruleset rs* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*normalize-rules-dnf rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
unfolding *normalize-rules-dnf-def2*
apply(*rule normalize-rules-match-list-semantics*)
apply (*metis matches-to-match-list-normalize*)
by *simp*

lemma *normalize-rules-dnf-correct*: *wf-ruleset* γ *p rs* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*normalize-rules-dnf rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
proof(*induction rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
thus *?case* (**is** *?goal*)
proof(*cases s*)
case *Decision* **thus** *?goal*
by(*simp add*: *Decision-approximating-bigstep-fun*)
next
case *Undecided*
from *Cons wf-rulesetD*(*2*) **have** *IH*: *approximating-bigstep-fun* γ *p* (*normalize-rules-dnf rs*) *s* = *approximating-bigstep-fun* γ *p rs s* **by** *fast*
from *Cons.prems* **have** *wf-ruleset* γ *p* [*r*] **and** *wf-ruleset* γ *p* (*normalize-rules-dnf* [*r*])
by(*auto dest*: *wf-rulesetD simp*: *wf-ruleset-normalize-rules-dnf*)
with *IH Undecided* **have**
approximating-bigstep-fun γ *p* (*normalize-rules-dnf rs*) (*approximating-bigstep-fun* γ *p* (*normalize-rules-dnf* [*r*]) *Undecided*) = *approximating-bigstep-fun* γ *p* (*r* # *rs*) *Undecided*
apply(*cases r*, *rename-tac m a*)
apply(*simp*)
apply(*case-tac a*)
apply(*simp-all add*: *normalize-match-correct Decision-approximating-bigstep-fun wf-ruleset-singleton*)
done
hence *approximating-bigstep-fun* γ *p* (*normalize-rules-dnf* [*r*] @ *normalize-rules-dnf rs*) *s* = *approximating-bigstep-fun* γ *p* (*r* # *rs*) *s*
using *Undecided* ‹*wf-ruleset* γ *p* [*r*]› ‹*wf-ruleset* γ *p* (*normalize-rules-dnf* [*r*])›

by(*simp add*: *approximating-bigstep-fun-seq-wf*)
thus *?goal* **using** *normalize-rules-fst normalize-rules-dnf-def2* **by** *metis*
qed
qed

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 reuslt is a list. Essentially, this is the disjunctive normal form.

lemma *normalized-nnf-match-normalize-match*: $\forall$ *m'* $\in$ *set* (*normalize-match* *m*). *normalized-nnf-match* *m'*
proof(*induction* *m* *arbitrary*: *rule*: *normalize-match.induct*)
case *4* **thus** *?case* **by** *fastforce*
qed (*simp-all*)

Example

lemma *normalize-match* (*MatchNot* (*MatchAnd* (*Match* *ip-src*) (*Match* *tcp*))) = [*MatchNot* (*Match* *ip-src*), *MatchNot* (*Match* *tcp*)] **by** *simp*

lemma *optimize-matches-normalized-nnf-match*: $\llbracket\forall$ *r* $\in$ *set* *rs*. *normalized-nnf-match* (*get-match* *r*); $\forall$ *m*. *normalized-nnf-match* *m* $\longrightarrow$ *normalized-nnf-match* (*f* *m*) $\rrbracket \Longrightarrow$
$\forall$ *r* $\in$ *set* (*optimize-matches* *f* *rs*). *normalized-nnf-match* (*get-match* *r*)
proof(*induction* *rs*)
case *Nil* **thus** *?case* **unfolding** *optimize-matches-def* **by** *simp*
next
case (*Cons* *r* *rs*)
from *Cons.IH* *Cons.prems* **have** *IH*: $\forall$ *r*$\in$*set* (*optimize-matches* *f* *rs*). *normalized-nnf-match* (*get-match* *r*) **by** *simp*
from *Cons.prems* **have** $\forall$ *r*$\in$*set* (*optimize-matches* *f* [*r*]). *normalized-nnf-match* (*get-match* *r*)
by(*simp* *add*: *optimize-matches-def*)
with *IH* **show** *?case* **by**(*simp* *add*: *optimize-matches-def*)
qed

lemma *normalize-rules-dnf-normalized-nnf-match*: $\forall$ *x* $\in$ *set* (*normalize-rules-dnf* *rs*). *normalized-nnf-match* (*get-match* *x*)
proof(*induction* *rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons* *r* *rs*) **thus** *?case* **using** *normalized-nnf-match-normalize-match* **by**(*cases* *r*) *fastforce*
qed

end
theory *Negation-Type-Matching*
imports *../Common/Negation-Type Matching-Ternary ../Datatype-Selectors Fixed-Action*
begin

19 Negation Type Matching

Transform a *'a negation-type list* to a *'a match-expr* via conjunction.

fun *alist-and* :: *'a negation-type list* ⇒ *'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*)

fun *negation-type-to-match-expr* :: *'a negation-type* ⇒ *'a match-expr* **where**
  negation-type-to-match-expr (*Pos e*) = (*Match e*) |
  negation-type-to-match-expr (*Neg e*) = (*MatchNot* (*Match e*))
lemma *alist-and-negation-type-to-match-expr*: *alist-and* (*n*#*es*) = *MatchAnd* (*negation-type-to-match-expr n*) (*alist-and es*)
  by(*cases n*, *simp-all*)

fun *negation-type-to-match-expr-f* :: (*'a* ⇒ *'b*) ⇒ *'a negation-type* ⇒ *'b match-expr*
where
  negation-type-to-match-expr-f f (*Pos a*) = *Match* (*f a*) |
  negation-type-to-match-expr-f f (*Neg a*) = *MatchNot* (*Match* (*f a*))

lemma *alist-and-append*: *matches* γ (*alist-and* (*l1* @ *l2*)) *a p* ⟷ *matches* γ (*MatchAnd* (*alist-and l1*) (*alist-and l2*)) *a p*
  proof(*induction l1*)
  case *Nil* **thus** *?case* **by** (*simp-all add*: *bunch-of-lemmata-about-matches*)
  next
  case (*Cons l l1*) **thus** *?case* **by** (*cases l*) (*simp-all add*: *bunch-of-lemmata-about-matches*)
  qed

fun *to-negation-type-nnf* :: *'a match-expr* ⇒ *'a negation-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* $\Longrightarrow$ *matches* γ (*alist-and* (*to-negation-type-nnf m*)) *a p* = *matches* γ *m a p*
proof(*induction m rule*: *to-negation-type-nnf.induct*)
qed(*simp-all add*: *bunch-of-lemmata-about-matches alist-and-append*)

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 γ *a p* ((*Pos x*)#*xs*) $\longleftrightarrow$ *matches* γ (*Match x*) *a p* $\wedge$ *nt-match-list* γ *a p xs* |
nt-match-list γ *a p* ((*Neg x*)#*xs*) $\longleftrightarrow$ *matches* γ (*MatchNot* (*Match x*)) *a p* $\wedge$ *nt-match-list* γ *a p xs*

lemma *nt-match-list-matches*: *nt-match-list* γ *a p l* $\longleftrightarrow$ *matches* γ (*alist-and l*) *a p*
apply(*induction l rule*: *alist-and.induct*)
apply(*case-tac* [!] γ)
apply(*simp-all add*: *bunch-of-lemmata-about-matches*)
done

lemma *nt-match-list-simp*: *nt-match-list* γ *a p ms* $\longleftrightarrow$
($\forall$ *m* $\in$ *set* (*getPos ms*). *matches* γ (*Match m*) *a p*) $\wedge$ ($\forall$ *m* $\in$ *set* (*getNeg ms*). *matches* γ (*MatchNot* (*Match m*)) *a p*)
proof(*induction* γ *a p ms rule*: *nt-match-list.induct*)
case *3* **thus** *?case* **by** *fastforce*
qed(*simp-all*)

lemma *matches-alist-and*: *matches* γ (*alist-and l*) *a p* $\longleftrightarrow$ ($\forall$ *m* $\in$ *set* (*getPos l*). *matches* γ (*Match m*) *a p*) $\wedge$ ($\forall$ *m* $\in$ *set* (*getNeg l*). *matches* γ (*MatchNot* (*Match m*)) *a p*)
using *nt-match-list-matches nt-match-list-simp* **by** *fast*

end
theory *Packet-Set-Impl*
imports *Fixed-Action Negation-Type-Matching ../Datatype-Selectors*
begin

20 Util: listprod

definition *listprod* :: *nat list* $\Rightarrow$ *nat* **where** *listprod as* $\equiv$ *foldr* (*op* $*$) *as 1*

lemma *listprod-append*[*simp*]: *listprod* (*as* @ *bs*) = *listprod as* $*$ *listprod bs*
apply(*induction as arbitrary*: *bs*)
apply(*simp-all add*: *listprod-def*)
done
lemma *listprod-simps* [*simp*]:

listprod [] = *1*
listprod (*x* # *xs*) = *x* ∗ *listprod xs*
by (*simp-all add*: *listprod-def*)
lemma *distinct as* $\Longrightarrow$ *listprod as* = $\prod$(*set as*)
by(*induction as*) *simp-all*

21 Executable Packet Set Representation

Recall: *alist-and* transforms *'a negation-type list* ⇒ *'a match-expr* and uses conjunction as connective.

Symbolic (executable) representation. inner is ∧, outer is ∨

datatype *'a packet-set* = *PacketSet* (*packet-set-repr*: ((*'a negation-type* × *action negation-type*) *list*) *list*)

Essentially, the *'a list list* structure represents a DNF. See `../Common/Negation_Type_DNF.thy` for a pure Boolean version (without matching).

definition *to-packet-set* :: *action* ⇒ *'a match-expr* ⇒ *'a packet-set* **where**
to-packet-set a m = *PacketSet* (*map* (*map* (λ*m'*. (*m'*,*Pos a*)) *o to-negation-type-nnf*) (*normalize-match m*))

fun *get-action* :: *action negation-type* ⇒ *action* **where**
get-action (*Pos a*) = *a* |
get-action (*Neg a*) = *a*

fun *get-action-sign* :: *action negation-type* ⇒ (*bool* ⇒ *bool*) **where**
get-action-sign (*Pos* -) = *id* |
get-action-sign (*Neg* -) = (λ*m*. ¬ *m*)

We collect all entries of the outer list. For the inner list, we request that a packet matches all the entries. A negated action means that the expression must not match. Recall: *matches* γ (*MatchNot m*) *a p* ≠ (¬ *matches* γ *m a p*), due to *TernaryUnknown*

definition *packet-set-to-set* :: (*'a*, *'packet*) *match-tac* ⇒ *'a packet-set* ⇒ *'packet set* **where**
packet-set-to-set γ *ps* ≡ $\bigcup$ *ms* ∈ *set* (*packet-set-repr ps*). {*p*. ∀ (*m*, *a*) ∈ *set ms*. *get-action-sign a* (*matches* γ (*negation-type-to-match-expr m*) (*get-action a*) *p*)}

lemma *packet-set-to-set-alt*: *packet-set-to-set* γ *ps* = ($\bigcup$ *ms* ∈ *set* (*packet-set-repr ps*).
{*p*. ∀ *m a*. (*m*, *a*) ∈ *set ms* ⟶ *get-action-sign a* (*matches* γ (*negation-type-to-match-expr m*) (*get-action a*) *p*)})
unfolding *packet-set-to-set-def*
by *fast*

We really have a disjunctive normal form

lemma *packet-set-to-set-alt2*: *packet-set-to-set* γ *ps* = ($\bigcup$ *ms* $\in$ *set* (*packet-set-repr ps*).
($\bigcap$ (*m*, *a*) $\in$ *set ms*. {*p*. *get-action-sign a* (*matches* γ (*negation-type-to-match-expr m*) (*get-action a*) *p*)}))
unfolding *packet-set-to-set-alt*
by *blast*

lemma *to-packet-set-correct*: *p* $\in$ *packet-set-to-set* γ (*to-packet-set a m*) $\longleftrightarrow$ *matches* γ *m a p*
apply(*simp add*: *to-packet-set-def packet-set-to-set-def*)
apply(*rule iffI*)
apply(*clarify*)
apply(*induction m rule*: *normalize-match.induct*)
apply(*simp-all add*: *bunch-of-lemmata-about-matches*)
apply *force*
apply (*metis matches-DeMorgan*)
apply(*induction m rule*: *normalize-match.induct*)
apply(*simp-all add*: *bunch-of-lemmata-about-matches*)
apply (*metis Un-iff*)
apply (*metis Un-iff matches-DeMorgan*)
done

lemma *to-packet-set-set*: *packet-set-to-set* γ (*to-packet-set a m*) = {*p*. *matches* γ *m a p*}
using *to-packet-set-correct* **by** *fast*

definition *packet-set-UNIV* :: *'a packet-set* **where**
packet-set-UNIV $\equiv$ *PacketSet* [[]]
lemma *packet-set-UNIV*: *packet-set-to-set* γ *packet-set-UNIV* = *UNIV*
by(*simp add*: *packet-set-UNIV-def packet-set-to-set-def*)

definition *packet-set-Empty* :: *'a packet-set* **where**
packet-set-Empty $\equiv$ *PacketSet* []
lemma *packet-set-Empty*: *packet-set-to-set* γ *packet-set-Empty* = {}
by(*simp add*: *packet-set-Empty-def packet-set-to-set-def*)

If the matching agrees for two actions, then the packet sets are also equal

lemma $\forall$ *p*. *matches* γ *m a1 p* $\longleftrightarrow$ *matches* γ *m a2 p* $\Longrightarrow$ *packet-set-to-set* γ (*to-packet-set a1 m*) = *packet-set-to-set* γ (*to-packet-set a2 m*)
apply(*subst*(*asm*) *to-packet-set-correct*[*symmetric*])+
apply *safe*
apply *simp-all*
done

21.0.1 Basic Set Operations

$\cap$

fun *packet-set-intersect* :: *'a packet-set* $\Rightarrow$ *'a packet-set* $\Rightarrow$ *'a packet-set* **where**

packet-set-intersect (*PacketSet olist1*) (*PacketSet olist2*) = *PacketSet* [*andlist1* @ *andlist2*. *andlist1* <− *olist1*, *andlist2* <− *olist2*]

lemma *packet-set-intersect* (*PacketSet* [[*a*,*b*], [*c*,*d*]]) (*PacketSet* [[*v*,*w*], [*x*,*y*]]) = *PacketSet* [[*a*, *b*, *v*, *w*], [*a*, *b*, *x*, *y*], [*c*, *d*, *v*, *w*], [*c*, *d*, *x*, *y*]] **by** *simp*

declare *packet-set-intersect.simps*[*simp del*]

lemma *packet-set-intersect-intersect*: *packet-set-to-set* γ (*packet-set-intersect P1 P2*) = *packet-set-to-set* γ *P1* $\cap$ *packet-set-to-set* γ *P2*
unfolding *packet-set-to-set-def*
apply(*cases P1*)
apply(*cases P2*)
apply(*simp*)
apply(*simp add*: *packet-set-intersect.simps*)
apply *blast*
done

lemma *packet-set-intersect-correct*: *packet-set-to-set* γ (*packet-set-intersect* (*to-packet-set a m1*) (*to-packet-set a m2*)) = *packet-set-to-set* γ (*to-packet-set a* (*MatchAnd m1 m2*))
apply(*simp add*: *to-packet-set-def packet-set-intersect.simps packet-set-to-set-alt*)

apply *safe*
apply *simp-all*
apply *blast+*
done

lemma *packet-set-intersect-correct'*: *p* $\in$ *packet-set-to-set* γ (*packet-set-intersect* (*to-packet-set a m1*) (*to-packet-set a m2*)) $\longleftrightarrow$ *matches* γ (*MatchAnd m1 m2*) *a p*
apply(*simp add*: *to-packet-set-correct*[*symmetric*])
using *packet-set-intersect-correct* **by** *fast*

The length of the result is the product of the input lengths

lemma *packet-set-intersetc-length*: *length* (*packet-set-repr* (*packet-set-intersect* (*PacketSet ass*) (*PacketSet bss*))) = *length ass* $*$ *length bss*
by(*induction ass*) (*simp-all add*: *packet-set-intersect.simps*)

$\cup$

fun *packet-set-union* :: *'a packet-set* $\Rightarrow$ *'a packet-set* $\Rightarrow$ *'a packet-set* **where**
packet-set-union (*PacketSet olist1*) (*PacketSet olist2*) = *PacketSet* (*olist1* @ *olist2*)
declare *packet-set-union.simps*[*simp del*]

lemma *packet-set-union-correct*: *packet-set-to-set* γ (*packet-set-union P1 P2*) = *packet-set-to-set* γ *P1* $\cup$ *packet-set-to-set* γ *P2*

unfolding *packet-set-to-set-def*
apply(*cases P1*)
apply(*cases P2*)
apply(*simp add*: *packet-set-union.simps*)
done

lemma *packet-set-append*:
packet-set-to-set γ (*PacketSet* (*p1* @ *p2*)) = *packet-set-to-set* γ (*PacketSet* *p1*) $\cup$ *packet-set-to-set* γ (*PacketSet p2*)
by(*simp add*: *packet-set-to-set-def*)
lemma *packet-set-cons*: *packet-set-to-set* γ (*PacketSet* (*a* # *p3*)) = *packet-set-to-set* γ (*PacketSet* [*a*]) $\cup$ *packet-set-to-set* γ (*PacketSet p3*)
by(*simp add*: *packet-set-to-set-def*)

—

fun *listprepend* :: *'a list* $\Rightarrow$ *'a list list* $\Rightarrow$ *'a list list* **where**
listprepend [] *ns* = [] |
listprepend (*a#as*) *ns* = (*map* (λ*xs*. *a#xs*) *ns*) @ (*listprepend as ns*)

The returned result of *listprepend* can get long.

lemma *listprepend-length*: *length* (*listprepend as bss*) = *length as* ∗ *length bss*
by(*induction as*) (*simp-all*)

lemma *packet-set-map-a-and*: *packet-set-to-set* γ (*PacketSet* (*map* (*op* # *a*) *ds*)) = *packet-set-to-set* γ (*PacketSet* [[*a*]]) $\cap$ *packet-set-to-set* γ (*PacketSet ds*)
apply(*induction ds*)
apply(*simp-all add*: *packet-set-to-set-def*)
apply(*case-tac a*)
apply(*simp-all*)
apply *blast+*
done
lemma *listprepend-correct*: *packet-set-to-set* γ (*PacketSet* (*listprepend as ds*)) = *packet-set-to-set* γ (*PacketSet* (*map* (λ*a*. [*a*]) *as*)) $\cap$ *packet-set-to-set* γ (*PacketSet* *ds*)
apply(*induction as arbitrary*:)
apply(*simp add*: *packet-set-to-set-alt*)
apply(*simp*)
apply(*rename-tac a as*)
apply(*simp add*: *packet-set-map-a-and packet-set-append*)

apply(*subst*(*2*) *packet-set-cons*)
by *blast*

lemma *packet-set-to-set-map-singleton*: *packet-set-to-set* γ (*PacketSet* (*map* (λ*a*. [*a*]) *as*)) = ($\bigcup$ *a* $\in$ *set as*. *packet-set-to-set* γ (*PacketSet* [[*a*]]))
by(*simp add*: *packet-set-to-set-alt*)

fun *invertt* :: (*'a negation-type* $\times$ *action negation-type*) $\Rightarrow$ (*'a negation-type* $\times$ *action negation-type*) **where**

invertt $(n,\ a) = (n,\ invert\ a)$

lemma *singleton-invertt*: *packet-set-to-set* γ (*PacketSet* [[*invertt n*]]) = − *packet-set-to-set* γ (*PacketSet* [[*n*]])
apply(*simp add*: *to-packet-set-def packet-set-intersect.simps packet-set-to-set-alt*)
apply(*case-tac n*, *rename-tac m a*)
apply(*simp*)
apply(*case-tac a*)
apply(*simp-all*)
apply *safe*
done

lemma *packet-set-to-set-map-singleton-invertt*:
packet-set-to-set γ (*PacketSet* (*map* (($\lambda a.$ [*a*]) $\circ$ *invertt*) *d*)) = − ($\bigcap$ $a \in$ *set d*. *packet-set-to-set* γ (*PacketSet* [[*a*]]))
apply(*induction d*)
apply(*simp*)
apply(*simp add*: *packet-set-to-set-alt*)
apply(*simp add*:)
apply(*subst*(*1*) *packet-set-cons*)
apply(*simp*)
apply(*simp add*: *packet-set-to-set-map-singleton singleton-invertt*)
done

fun *packet-set-not-internal* :: ('*a negation-type* × *action negation-type*) *list list* ⇒ ('*a negation-type* × *action negation-type*) *list list* **where**
packet-set-not-internal [] = [[]] |
packet-set-not-internal (*ns*#*nss*) = *listprepend* (*map invertt ns*) (*packet-set-not-internal nss*)

lemma *packet-set-not-internal-length*: *length* (*packet-set-not-internal ass*) = *listprod* ([*length n*. *n* <− *ass*])
by(*induction ass*) (*simp-all add*: *listprepend-length*)

lemma *packet-set-not-internal-correct*: *packet-set-to-set* γ (*PacketSet* (*packet-set-not-internal d*)) = − *packet-set-to-set* γ (*PacketSet d*)
apply(*induction d*)
apply(*simp add*: *packet-set-to-set-alt*)
apply(*rename-tac d ds*)
apply(*simp add*:)
apply(*simp add*: *listprepend-correct*)
apply(*simp add*: *packet-set-to-set-map-singleton-invertt*)
apply(*simp add*: *packet-set-to-set-alt*)
by *blast*

fun *packet-set-not* :: '*a packet-set* ⇒ '*a packet-set* **where**
packet-set-not (*PacketSet ps*) = *PacketSet* (*packet-set-not-internal ps*)
declare *packet-set-not.simps*[*simp del*]

The length of the result of *packet-set-not* is the multiplication over the length of all the inner sets. Warning: gets huge! See *length* (*packet-set-not-internal ?ass*) = *Packet-Set-Impl.listprod* (*map length ?ass*)

lemma *packet-set-not-correct*: *packet-set-to-set* γ (*packet-set-not P*) = − *packet-set-to-set* γ *P*
  apply(*cases P*)
  apply(*simp*)
  apply(*simp add*: *packet-set-not.simps*)
  apply(*simp add*: *packet-set-not-internal-correct*)
  done

21.0.2 Derived Operations

definition *packet-set-constrain* :: *action* ⇒ *'a match-expr* ⇒ *'a packet-set* ⇒ *'a packet-set* **where**
  packet-set-constrain a m ns = *packet-set-intersect ns* (*to-packet-set a m*)

theorem *packet-set-constrain-correct*: *packet-set-to-set* γ (*packet-set-constrain a m P*) = {*p* ∈ *packet-set-to-set* γ *P*. *matches* γ *m a p*}
unfolding *packet-set-constrain-def*
unfolding *packet-set-intersect-intersect*
unfolding *to-packet-set-set*
by *blast*

Warning: result gets huge

definition *packet-set-constrain-not* :: *action* ⇒ *'a match-expr* ⇒ *'a packet-set* ⇒ *'a packet-set* **where**
  packet-set-constrain-not a m ns = *packet-set-intersect ns* (*packet-set-not* (*to-packet-set a m*))

theorem *packet-set-constrain-not-correct*: *packet-set-to-set* γ (*packet-set-constrain-not a m P*) = {*p* ∈ *packet-set-to-set* γ *P*. ¬ *matches* γ *m a p*}
unfolding *packet-set-constrain-not-def*
unfolding *packet-set-intersect-intersect*
unfolding *packet-set-not-correct*
unfolding *to-packet-set-set*
by *blast*

21.0.3 Optimizing

fun *packet-set-opt1* :: *'a packet-set* ⇒ *'a packet-set* **where**
  packet-set-opt1 (*PacketSet ps*) = *PacketSet* (*map remdups* (*remdups ps*))
declare *packet-set-opt1.simps*[*simp del*]

lemma *packet-set-opt1-correct*: *packet-set-to-set* γ (*packet-set-opt1 ps*) = *packet-set-to-set* γ *ps*
  by(*cases ps*) (*simp add*: *packet-set-to-set-alt packet-set-opt1.simps*)

fun *packet-set-opt2-internal* :: $(('a\ negation\text{-}type \times action\ negation\text{-}type)\ list)\ list \Rightarrow (('a\ negation\text{-}type \times action\ negation\text{-}type)\ list)\ list$ **where**
packet-set-opt2-internal [] = [] |

packet-set-opt2-internal ([]#ps) = [[]] |

packet-set-opt2-internal (as#ps) = as# (*if length as* $\leq$ *5 then packet-set-opt2-internal* ((*filter* (λ*ass.* $\neg$ *set as* $\subseteq$ *set ass*) *ps*)) *else packet-set-opt2-internal ps*)

lemma *packet-set-opt2-internal-correct*: *packet-set-to-set* γ (*PacketSet* (*packet-set-opt2-internal ps*)) = *packet-set-to-set* γ (*PacketSet ps*)
apply(*induction ps rule*:*packet-set-opt2-internal.induct*)
apply(*simp-all add*: *packet-set-UNIV*)
apply(*simp add*: *packet-set-to-set-alt*)
apply(*simp add*: *packet-set-to-set-alt*)
apply(*safe*)[*1*]
apply(*simp-all*)
apply *blast+*

done

export-code *packet-set-opt2-internal* **in** *SML*

fun *packet-set-opt2* :: *'a packet-set* $\Rightarrow$ *'a packet-set* **where**
packet-set-opt2 (*PacketSet ps*) = *PacketSet* (*packet-set-opt2-internal ps*)
declare *packet-set-opt2.simps*[*simp del*]

lemma *packet-set-opt2-correct*: *packet-set-to-set* γ (*packet-set-opt2 ps*) = *packet-set-to-set* γ *ps*
by(*cases ps*) (*simp add*: *packet-set-opt2.simps packet-set-opt2-internal-correct*)

If we sort by length, we will hopefully get better results when applying *packet-set-opt2*.

fun *packet-set-opt3* :: *'a packet-set* $\Rightarrow$ *'a packet-set* **where**
packet-set-opt3 (*PacketSet ps*) = *PacketSet* (*sort-key* (λ*p. length p*) *ps*)
declare *packet-set-opt3.simps*[*simp del*]
lemma *packet-set-opt3-correct*: *packet-set-to-set* γ (*packet-set-opt3 ps*) = *packet-set-to-set* γ *ps*
by(*cases ps*) (*simp add*: *packet-set-opt3.simps packet-set-to-set-alt*)

fun *packet-set-opt4-internal-internal* :: $(('a\ negation\text{-}type \times action\ negation\text{-}type)\ list) \Rightarrow bool$ **where**

packet-set-opt4-internal-internal cs = (∀ (*m*, *a*) ∈ *set cs*. (*m*, *invert a*) ∉ *set cs*)

fun *packet-set-opt4* :: *'a packet-set* ⇒ *'a packet-set* **where**
packet-set-opt4 (*PacketSet ps*) = *PacketSet* (*filter packet-set-opt4-internal-internal ps*)

declare *packet-set-opt4.simps*[*simp del*]

lemma *packet-set-opt4-internal-internal-helper*: **assumes**
∀ *m a*. (*m*, *a*) ∈ *set xb* ⟶ *get-action-sign a* (*matches* γ (*negation-type-to-match-expr m*) (*get-action a*) *xa*)
shows ∀ (*m*, *a*)∈*set xb*. (*m*, *invert a*) ∉ *set xb*
proof(*clarify*)
fix *a b*
assume *a1*: (*a*, *b*) ∈ *set xb* **and** *a2*: (*a*, *invert b*) ∈ *set xb*
from *assms a1* **have** *1*: *get-action-sign b* (*matches* γ (*negation-type-to-match-expr a*) (*get-action b*) *xa*) **by** *simp*
from *assms a2* **have** *2*: *get-action-sign* (*invert b*) (*matches* γ (*negation-type-to-match-expr a*) (*get-action* (*invert b*)) *xa*) **by** *simp*
from *1 2* **show** *False*
by(*cases b*) (*simp-all*)
qed

lemma *packet-set-opt4-correct*: *packet-set-to-set* γ (*packet-set-opt4 ps*) = *packet-set-to-set* γ *ps*
apply(*cases ps*, *clarify*)
apply(*simp add*: *packet-set-opt4.simps packet-set-to-set-alt*)
apply(*rule*)
apply *blast*
apply(*clarify*)
apply(*simp*)
apply(*rule-tac x=xb* **in** *exI*)
apply(*simp*)
using *packet-set-opt4-internal-internal-helper* **by** *fast*

definition *packet-set-opt* :: *'a packet-set* ⇒ *'a packet-set* **where**
packet-set-opt ps = *packet-set-opt1* (*packet-set-opt2* (*packet-set-opt3* (*packet-set-opt4 ps*)))

lemma *packet-set-opt-correct*: *packet-set-to-set* γ (*packet-set-opt ps*) = *packet-set-to-set* γ *ps*
using *packet-set-opt-def packet-set-opt2-correct packet-set-opt3-correct packet-set-opt4-correct packet-set-opt1-correct* **by** *metis*

21.1 Conjunction Normal Form Packet Set

datatype *'a packet-set-cnf* = *PacketSetCNF* (*packet-set-repr-cnf*: ((*'a negation-type* × *action negation-type*) *list*) *list*)

lemma ¬ ((*a* ∧ *b*) ∨ (*c* ∧ *d*)) ⟷ (¬*a* ∨ ¬*b*) ∧ (¬*c* ∨ ¬ *d*) **by** *blast*

lemma $\neg ((a \vee b) \wedge (c \vee d)) \longleftrightarrow (\neg a \wedge \neg b) \vee (\neg c \wedge \neg d)$ **by** *blast*

definition *packet-set-cnf-to-set* :: ('*a*, '*packet*) *match-tac* ⇒ '*a packet-set-cnf* ⇒ '*packet set* **where**
packet-set-cnf-to-set γ *ps* ≡ ($\bigcap$ *ms* ∈ *set* (*packet-set-repr-cnf ps*).
($\bigcup$ (*m*, *a*) ∈ *set ms*. {*p*. *get-action-sign a* (*matches* γ (*negation-type-to-match-expr m*) (*get-action a*) *p*)}))

Inverting a '*a packet-set* and returning '*a packet-set-cnf* is very efficient!

fun *packet-set-not-to-cnf* :: '*a packet-set* ⇒ '*a packet-set-cnf* **where**
packet-set-not-to-cnf (*PacketSet ps*) = *PacketSetCNF* (*map* (λ*a*. *map invertt a*) *ps*)
declare *packet-set-not-to-cnf*.*simps*[*simp del*]

lemma *helper*: (*case invertt x of* (*m*, *a*) ⇒ {*p*. *get-action-sign a* (*matches* γ (*negation-type-to-match-expr m*) (*Packet-Set-Impl*.*get-action a*) *p*)}) =
(− (*case x of* (*m*, *a*) ⇒ {*p*. *get-action-sign a* (*matches* γ (*negation-type-to-match-expr m*) (*Packet-Set-Impl*.*get-action a*) *p*)}))
apply(*case-tac x*)
apply(*simp*)
apply(*case-tac b*)
apply(*simp-all*)
apply *safe*
done
lemma *packet-set-not-to-cnf-correct*: *packet-set-cnf-to-set* γ (*packet-set-not-to-cnf P*) = − *packet-set-to-set* γ *P*
apply(*cases P*)
apply(*simp add*: *packet-set-not-to-cnf*.*simps packet-set-cnf-to-set-def packet-set-to-set-alt2*)
apply(*subst helper*)
by *simp*

fun *packet-set-cnf-not-to-dnf* :: '*a packet-set-cnf* ⇒ '*a packet-set* **where**
packet-set-cnf-not-to-dnf (*PacketSetCNF ps*) = *PacketSet* (*map* (λ*a*. *map invertt a*) *ps*)
declare *packet-set-cnf-not-to-dnf*.*simps*[*simp del*]
lemma *packet-set-cnf-not-to-dnf-correct*: *packet-set-to-set* γ (*packet-set-cnf-not-to-dnf P*) = − *packet-set-cnf-to-set* γ *P*
apply(*cases P*)
apply(*simp add*: *packet-set-cnf-not-to-dnf*.*simps packet-set-cnf-to-set-def packet-set-to-set-alt2*)
apply(*subst helper*)
by *simp*

Also, intersection is highly efficient in CNF

fun *packet-set-cnf-intersect* :: '*a packet-set-cnf* ⇒ '*a packet-set-cnf* ⇒ '*a packet-set-cnf* **where**
packet-set-cnf-intersect (*PacketSetCNF ps1*) (*PacketSetCNF ps2*) = *PacketSetCNF* (*ps1* @ *ps2*)
declare *packet-set-cnf-intersect*.*simps*[*simp del*]

lemma *packet-set-cnf-intersect-correct*: *packet-set-cnf-to-set* γ (*packet-set-cnf-intersect* *P1 P2*) = *packet-set-cnf-to-set* γ *P1* $\cap$ *packet-set-cnf-to-set* γ *P2*
apply(*case-tac P1*)
apply(*case-tac P2*)
apply(*simp add*: *packet-set-cnf-to-set-def packet-set-cnf-intersect.simps*)
apply(*safe*)
apply(*simp-all*)
done

Optimizing

fun *packet-set-cnf-opt1* :: *'a packet-set-cnf* $\Rightarrow$ *'a packet-set-cnf* **where**
packet-set-cnf-opt1 (*PacketSetCNF ps*) = *PacketSetCNF* (*map remdups* (*remdups* *ps*))
declare *packet-set-cnf-opt1.simps*[*simp del*]

lemma *packet-set-cnf-opt1-correct*: *packet-set-cnf-to-set* γ (*packet-set-cnf-opt1* *ps*) = *packet-set-cnf-to-set* γ *ps*
by(*cases ps*) (*simp add*: *packet-set-cnf-to-set-def packet-set-cnf-opt1.simps*)

fun *packet-set-cnf-opt2-internal* :: ((*'a negation-type* $\times$ *action negation-type*) *list*) *list* $\Rightarrow$ ((*'a negation-type* $\times$ *action negation-type*) *list*) *list* **where**
packet-set-cnf-opt2-internal [] = [] |
packet-set-cnf-opt2-internal ([]#*ps*) = [[]] |

packet-set-cnf-opt2-internal (*as*#*ps*) = (*as*#(*filter* (λ*ass*. $\neg$ *set as* $\subseteq$ *set ass*) *ps*))

lemma *packet-set-cnf-opt2-internal-correct*: *packet-set-cnf-to-set* γ (*PacketSetCNF* (*packet-set-cnf-opt2-internal ps*)) = *packet-set-cnf-to-set* γ (*PacketSetCNF ps*)
apply(*induction ps rule*:*packet-set-cnf-opt2-internal.induct*)
apply(*simp-all add*: *packet-set-cnf-to-set-def*)
by *blast*
fun *packet-set-cnf-opt2* :: *'a packet-set-cnf* $\Rightarrow$ *'a packet-set-cnf* **where**
packet-set-cnf-opt2 (*PacketSetCNF ps*) = *PacketSetCNF* (*packet-set-cnf-opt2-internal* *ps*)
declare *packet-set-cnf-opt2.simps*[*simp del*]

lemma *packet-set-cnf-opt2-correct*: *packet-set-cnf-to-set* γ (*packet-set-cnf-opt2* *ps*) = *packet-set-cnf-to-set* γ *ps*
by(*cases ps*) (*simp add*: *packet-set-cnf-opt2.simps packet-set-cnf-opt2-internal-correct*)

fun *packet-set-cnf-opt3* :: *'a packet-set-cnf* $\Rightarrow$ *'a packet-set-cnf* **where**
packet-set-cnf-opt3 (*PacketSetCNF ps*) = *PacketSetCNF* (*sort-key* (λ*p*. *length* *p*) *ps*)
declare *packet-set-cnf-opt3.simps*[*simp del*]
lemma *packet-set-cnf-opt3-correct*: *packet-set-cnf-to-set* γ (*packet-set-cnf-opt3* *ps*) = *packet-set-cnf-to-set* γ *ps*
by(*cases ps*) (*simp add*: *packet-set-cnf-opt3.simps packet-set-cnf-to-set-def*)

definition *packet-set-cnf-opt* :: *'a packet-set-cnf* ⇒ *'a packet-set-cnf* **where**
packet-set-cnf-opt ps = *packet-set-cnf-opt1* (*packet-set-cnf-opt2* (*packet-set-cnf-opt3* (*ps*)))

lemma *packet-set-cnf-opt-correct*: *packet-set-cnf-to-set* γ (*packet-set-cnf-opt ps*) = *packet-set-cnf-to-set* γ *ps*
using *packet-set-cnf-opt-def packet-set-cnf-opt2-correct packet-set-cnf-opt3-correct packet-set-cnf-opt1-correct* **by** *metis*

hide-const (**open**) *get-action get-action-sign packet-set-repr packet-set-repr-cnf*

end
theory *Packet-Set*
imports *Packet-Set-Impl*
begin

22 Packet Set

An explicit representation of sets of packets allowed/denied by a firewall. Very work in progress, such pre-alpha, wow. Probably everything here wants a simple ruleset.

22.1 The set of all accepted packets

Collect all packets which are allowed by the firewall.

fun *collect-allow* :: (*'a*, *'p*) *match-tac* ⇒ *'a rule list* ⇒ *'p set* ⇒ *'p set* **where**
collect-allow - [] *P* = {} |
collect-allow γ ((*Rule m Accept*)#*rs*) *P* = {*p* ∈ *P*. *matches* γ *m Accept p*} ∪ (*collect-allow* γ *rs* {*p* ∈ *P*. ¬ *matches* γ *m Accept p*}) |
collect-allow γ ((*Rule m Drop*)#*rs*) *P* = (*collect-allow* γ *rs* {*p* ∈ *P*. ¬ *matches* γ *m Drop p*})

lemma *collect-allow-subset*: *simple-ruleset rs* ⟹ *collect-allow* γ *rs P* ⊆ *P*
apply(*induction rs arbitrary*: *P*)
apply(*simp*)
apply(*rename-tac r rs P*)
apply(*case-tac r*, *rename-tac m a*)
apply(*case-tac a*)
apply(*simp-all add*: *simple-ruleset-def*)
apply(*fast*)
apply *blast*
done

lemma *collect-allow-sound*: *simple-ruleset rs* $\Longrightarrow$ *p* $\in$ *collect-allow* γ *rs P* $\Longrightarrow$ *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow*
proof(*induction rs arbitrary*: *P*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
from *Cons* **obtain** *m a* **where** *r*: *r* = *Rule m a* **by** (*cases r*) *simp*
from *Cons.prems* **have** *simple-rs*: *simple-ruleset rs* **by** (*simp add*: *r simple-ruleset-def*)
from *Cons.prems r* **have** *a-cases*: *a* = *Accept* $\vee$ *a* = *Drop* **by** (*simp add*: *r simple-ruleset-def*)
show *?case* (**is** *?goal*)
proof(*cases a*)
case *Accept*
from *Accept Cons.IH*[**where** *P*=$\{p \in P.\ \neg$ *matches* γ *m Accept p*$\}$] *simple-rs* **have** *IH*:
p $\in$ *collect-allow* γ *rs* $\{p \in P.\ \neg$ *matches* γ *m Accept p*$\}$ $\Longrightarrow$ *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow* **by** *simp*
from *Accept Cons.prems* **have** (*p* $\in$ *P* $\wedge$ *matches* γ *m Accept p*) $\vee$ *p* $\in$ *collect-allow* γ *rs* $\{p \in P.\ \neg$ *matches* γ *m Accept p*$\}$
by(*simp add*: *r*)
with *Accept* **show** *?goal*
apply −
apply(*erule disjE*)
apply(*simp add*: *r*)
apply(*simp add*: *r*)
using *IH* **by** *blast*
next
case *Drop*
from *Drop Cons.prems* **have** *p* $\in$ *collect-allow* γ *rs* $\{p \in P.\ \neg$ *matches* γ *m Drop p*$\}$
by(*simp add*: *r*)
with *Cons.IH simple-rs* **have** *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow* **by** *simp*
with *Cons* **show** *?goal*
apply(*simp add*: *r Drop del*: *approximating-bigstep-fun.simps*)
apply(*simp*)
using *collect-allow-subset*[*OF simple-rs*] **by** *fast*
qed(*insert a-cases*, *simp-all*)
qed

lemma *collect-allow-complete*: *simple-ruleset rs* $\Longrightarrow$ *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow* $\Longrightarrow$ *p* $\in$ *P* $\Longrightarrow$ *p* $\in$ *collect-allow* γ *rs P*
proof(*induction rs arbitrary*: *P*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)

from *Cons* **obtain** *m a* **where** *r*: *r* = *Rule m a* **by** (*cases r*) *simp*
from *Cons.prems* **have** *simple-rs*: *simple-ruleset rs* **by** (*simp add*: *r simple-ruleset-def*)
from *Cons.prems r* **have** *a-cases*: *a* = *Accept* ∨ *a* = *Drop* **by** (*simp add*: *r simple-ruleset-def*)
show *?case* (**is** *?goal*)
proof(*cases a*)
case *Accept*
from *Accept Cons.IH simple-rs* **have** *IH*: ∀ *P*. *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow* ⟶ *p* ∈ *P* ⟶ *p* ∈ *collect-allow* γ *rs P* **by** *simp*
with *Accept Cons.prems* **show** *?goal*
apply(*cases matches* γ *m Accept p*)
apply(*simp add*: *r*)
apply(*simp add*: *r*)
done
next
case *Drop*
with *Cons* **show** *?goal*
apply(*case-tac matches* γ *m Drop p*)
apply(*simp add*: *r*)
apply(*simp add*: *r simple-rs*)
done
qed(*insert a-cases*, *simp-all*)
qed

theorem *collect-allow-sound-complete*: *simple-ruleset rs* ⟹ {*p*. *p* ∈ *collect-allow* γ *rs UNIV*} = {*p*. *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow*}
apply(*safe*)
using *collect-allow-sound*[**where** *P*=*UNIV*] **apply** *fast*
using *collect-allow-complete*[**where** *P*=*UNIV*] **by** *fast*

the complement of the allowed packets

fun *collect-allow-compl* :: ('*a*, '*p*) *match-tac* ⇒ '*a rule list* ⇒ '*p set* ⇒ '*p set* **where**
collect-allow-compl - [] *P* = *UNIV* |
collect-allow-compl γ ((*Rule m Accept*)#*rs*) *P* = (*P* ∪ {*p*. ¬*matches* γ *m Accept p*}) ∩ (*collect-allow-compl* γ *rs* (*P* ∪ {*p*. *matches* γ *m Accept p*})) |
collect-allow-compl γ ((*Rule m Drop*)#*rs*) *P* = (*collect-allow-compl* γ *rs* (*P* ∪ {*p*. *matches* γ *m Drop p*}))

lemma *collect-allow-compl-correct*: *simple-ruleset rs* ⟹ (− *collect-allow-compl* γ *rs* (− *P*)) = *collect-allow* γ *rs P*
proof(*induction* γ *rs P arbitrary*: *P rule*: *collect-allow.induct*)
case *1* **thus** *?case* **by** *simp*
next
case (*2* γ *r rs*)
have *set-simp1*: − {*p* ∈ *P*. ¬ *matches* γ *r Accept p*} = − *P* ∪ {*p*. *matches*

γ *r Accept p*} **by** *blast*

from *2* **have** *IH*: $\bigwedge$*P*. − *collect-allow-compl* γ *rs* (− *P*) = *collect-allow* γ *rs* *P* **using** *simple-ruleset-tail* **by** *blast*

from *IH*[**where** *P*={*p* ∈ *P*. ¬ *matches* γ *r Accept p*}] *set-simp1* **have** − *collect-allow-compl* γ *rs* (− *P* ∪ *Collect* (*matches* γ *r Accept*)) = *collect-allow* γ *rs* {*p* ∈ *P*. ¬ *matches* γ *r Accept p*} **by** *simp*

thus *?case* **by** *auto*

next

case (*3* γ *r rs*)

have *set-simp1*: − {*p* ∈ *P*. ¬ *matches* γ *r Drop p*} = − *P* ∪ {*p*. *matches* γ *r Drop p*} **by** *blast*

from *3* **have** *IH*: $\bigwedge$*P*. − *collect-allow-compl* γ *rs* (− *P*) = *collect-allow* γ *rs* *P* **using** *simple-ruleset-tail* **by** *blast*

from *IH*[**where** *P*={*p* ∈ *P*. ¬ *matches* γ *r Drop p*}] *set-simp1* **have** − *collect-allow-compl* γ *rs* (− *P* ∪ *Collect* (*matches* γ *r Drop*)) = *collect-allow* γ *rs* {*p* ∈ *P*. ¬ *matches* γ *r Drop p*} **by** *simp*

thus *?case* **by** *auto*

qed(*simp-all add*: *simple-ruleset-def*)

22.2 The set of all dropped packets

Collect all packets which are denied by the firewall.

fun *collect-deny* :: (′*a*, ′*p*) *match-tac* ⇒ ′*a rule list* ⇒ ′*p set* ⇒ ′*p set* **where**
collect-deny - [] *P* = {} |
collect-deny γ ((*Rule m Drop*)#*rs*) *P* = {*p* ∈ *P*. *matches* γ *m Drop p*} ∪ (*collect-deny* γ *rs* {*p* ∈ *P*. ¬ *matches* γ *m Drop p*}) |
collect-deny γ ((*Rule m Accept*)#*rs*) *P* = (*collect-deny* γ *rs* {*p* ∈ *P*. ¬ *matches* γ *m Accept p*})

lemma *collect-deny-subset*: *simple-ruleset rs* ⟹ *collect-deny* γ *rs P* ⊆ *P*
apply(*induction rs arbitrary*: *P*)
apply(*simp*)
apply(*rename-tac r rs P*)
apply(*case-tac r*, *rename-tac m a*)
apply(*case-tac a*)
apply(*simp-all add*: *simple-ruleset-def*)
apply(*fast*)
apply *blast*
done

lemma *collect-deny-sound*: *simple-ruleset rs* ⟹ *p* ∈ *collect-deny* γ *rs P* ⟹ *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalDeny*
proof(*induction rs arbitrary*: *P*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
from *Cons* **obtain** *m a* **where** *r*: *r* = *Rule m a* **by** (*cases r*) *simp*
from *Cons.prems* **have** *simple-rs*: *simple-ruleset rs* **by** (*simp add*: *r simple-ruleset-def*)

from *Cons.prems r* **have** *a-cases*: $a = Accept \lor a = Drop$ **by** (*simp add*: *r simple-ruleset-def*)
show *?case* (**is** *?goal*)
proof(*cases a*)
case *Drop*
from *Drop Cons.IH*[**where** $P=\{p \in P.\ \neg$ *matches* γ *m Drop p*$\}$] *simple-rs* **have** *IH*:
$p \in$ *collect-deny* γ *rs* $\{p \in P.\ \neg$ *matches* γ *m Drop p*$\} \Longrightarrow$ *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalDeny* **by** *simp*
from *Drop Cons.prems* **have** ($p \in P \land$ *matches* γ *m Drop p*) $\lor\ p \in$ *collect-deny* γ *rs* $\{p \in P.\ \neg$ *matches* γ *m Drop p*$\}$
by(*simp add*: *r*)
with *Drop* **show** *?goal*
apply −
apply(*erule disjE*)
apply(*simp add*: *r*)
apply(*simp add*: *r*)
using *IH* **by** *blast*
next
case *Accept*
from *Accept Cons.prems* **have** $p \in$ *collect-deny* γ *rs* $\{p \in P.\ \neg$ *matches* γ *m Accept p*$\}$
by(*simp add*: *r*)
with *Cons.IH simple-rs* **have** *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalDeny* **by** *simp*
with *Cons* **show** *?goal*
apply(*simp add*: *r Accept del*: *approximating-bigstep-fun.simps*)
apply(*simp*)
using *collect-deny-subset*[*OF simple-rs*] **by** *fast*
qed(*insert a-cases*, *simp-all*)
qed

lemma *collect-deny-complete*: *simple-ruleset rs* $\Longrightarrow$ *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalDeny* $\Longrightarrow p \in P \Longrightarrow p \in$ *collect-deny* γ *rs P*
proof(*induction rs arbitrary*: *P*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
from *Cons* **obtain** *m a* **where** *r*: $r =$ *Rule m a* **by** (*cases r*) *simp*
from *Cons.prems* **have** *simple-rs*: *simple-ruleset rs* **by** (*simp add*: *r simple-ruleset-def*)
from *Cons.prems r* **have** *a-cases*: $a = Accept \lor a = Drop$ **by** (*simp add*: *r simple-ruleset-def*)
show *?case* (**is** *?goal*)
proof(*cases a*)
case *Accept*
from *Accept Cons.IH simple-rs* **have** *IH*: $\forall P.$ *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalDeny* $\longrightarrow p \in P \longrightarrow p \in$ *collect-deny* γ *rs P* **by** *simp*

with *Accept Cons.prems* **show** *?goal*
apply(*cases matches* γ *m Accept p*)
apply(*simp add*: *r*)
apply(*simp add*: *r*)
done
next
case *Drop*
with *Cons* **show** *?goal*
apply(*case-tac matches* γ *m Drop p*)
apply(*simp add*: *r*)
apply(*simp add*: *r simple-rs*)
done
qed(*insert a-cases*, *simp-all*)
qed

theorem *collect-deny-sound-complete*: *simple-ruleset rs* $\Longrightarrow$ $\{p.\ p \in$ *collect-deny* γ *rs UNIV*$\}$ = $\{p.$ *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalDeny*$\}$
apply(*safe*)
using *collect-deny-sound*[**where** *P*=*UNIV*] **apply** *fast*
using *collect-deny-complete*[**where** *P*=*UNIV*] **by** *fast*

the complement of the denied packets

fun *collect-deny-compl* :: ('*a*, '*p*) *match-tac* $\Rightarrow$ '*a rule list* $\Rightarrow$ '*p set* $\Rightarrow$ '*p set* **where**
collect-deny-compl - [] *P* = *UNIV* |
collect-deny-compl γ ((*Rule m Drop*)#*rs*) *P* = (*P* $\cup$ $\{p.\ \neg$*matches* γ *m Drop p*$\}$) $\cap$ (*collect-deny-compl* γ *rs* (*P* $\cup$ $\{p.$ *matches* γ *m Drop p*$\}$)) |
collect-deny-compl γ ((*Rule m Accept*)#*rs*) *P* = (*collect-deny-compl* γ *rs* (*P* $\cup$ $\{p.$ *matches* γ *m Accept p*$\}$))

lemma *collect-deny-compl-correct*: *simple-ruleset rs* $\Longrightarrow$ ($-$ *collect-deny-compl* γ *rs* ($-$ *P*)) = *collect-deny* γ *rs P*
proof(*induction* γ *rs P arbitrary*: *P rule*: *collect-deny.induct*)
case *1* **thus** *?case* **by** *simp*
next
case (*3* γ *r rs*)
have *set-simp1*: $-$ $\{p \in P.\ \neg$ *matches* γ *r Accept p*$\}$ = $-$ *P* $\cup$ $\{p.$ *matches* γ *r Accept p*$\}$ **by** *blast*
from *3* **have** *IH*: $\bigwedge P.$ $-$ *collect-deny-compl* γ *rs* ($-$ *P*) = *collect-deny* γ *rs P* **using** *simple-ruleset-tail* **by** *blast*
from *IH*[**where** *P*=$\{p \in P.\ \neg$ *matches* γ *r Accept p*$\}$] *set-simp1* **have**
$-$ *collect-deny-compl* γ *rs* ($-$ *P* $\cup$ *Collect* (*matches* γ *r Accept*)) = *collect-deny* γ *rs* $\{p \in P.\ \neg$ *matches* γ *r Accept p*$\}$ **by** *simp*
thus *?case* **by** *auto*
next
case (*2* γ *r rs*)
have *set-simp1*: $-$ $\{p \in P.\ \neg$ *matches* γ *r Drop p*$\}$ = $-$ *P* $\cup$ $\{p.$ *matches* γ

r Drop p} **by** *blast*

from *2* **have** *IH*: $\bigwedge P$. − *collect-deny-compl* γ *rs* (− *P*) = *collect-deny* γ *rs* *P* **using** *simple-ruleset-tail* **by** *blast*

from *IH*[**where** *P*={*p* ∈ *P*. ¬ *matches* γ *r Drop p*}] *set-simp1* **have**

− *collect-deny-compl* γ *rs* (− *P* ∪ *Collect* (*matches* γ *r Drop*)) = *collect-deny* γ *rs* {*p* ∈ *P*. ¬ *matches* γ *r Drop p*} **by** *simp*

thus *?case* **by** *auto*

qed(*simp-all add*: *simple-ruleset-def*)

22.3 Rulesets with default rules

definition *has-default* :: *'a rule list* ⇒ *bool* **where**

has-default rs ≡ *length rs* > *0* ∧ ((*last rs* = *Rule MatchAny Accept*) ∨ (*last rs* = *Rule MatchAny Drop*))

lemma *has-default-UNIV*: *good-ruleset rs* ⟹ *has-default rs* ⟹

{*p*. *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow*} ∪ {*p*. *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalDeny*} = *UNIV*

apply(*induction rs*)
apply(*simp add*: *has-default-def*)
apply(*rename-tac r rs*)
apply(*simp add*: *has-default-def good-ruleset-tail split*: *split-if-asm*)
apply(*elim disjE*)
apply(*simp add*: *bunch-of-lemmata-about-matches*)
apply(*simp add*: *bunch-of-lemmata-about-matches*)
apply(*case-tac r*, *rename-tac m a*)
apply(*case-tac a*)
apply(*auto simp*: *good-ruleset-def*)
done

lemma *allow-set-by-collect-deny-compl*: **assumes** *simple-ruleset rs* **and** *has-default rs*

shows *collect-deny-compl* γ *rs* {} = {*p*. *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow*}

proof −

from *assms* **have** *univ*: {*p*. *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow*} ∪ {*p*. *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalDeny*} = *UNIV*

using *simple-imp-good-ruleset has-default-UNIV* **by** *fast*

from *assms*(*1*) *collect-deny-compl-correct*[**where** *P*=*UNIV*] **have** *collect-deny-compl* γ *rs* {} = − *collect-deny* γ *rs UNIV* **by** *fastforce*

moreover with *collect-deny-sound-complete assms*(*1*) **have** ... = − {*p*. *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalDeny*} **by** *fast*

ultimately show *?thesis* **using** *univ* **by** *fastforce*

qed

lemma *deny-set-by-collect-allow-compl*: **assumes** *simple-ruleset rs* **and** *has-default rs*

shows *collect-allow-compl* γ *rs* {} = {*p*. *approximating-bigstep-fun* γ *p rs Un-*

decided = *Decision FinalDeny*}
proof −
from *assms* **have** *univ*: {*p*. *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow*} ∪ {*p*. *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalDeny*} = *UNIV*
using *simple-imp-good-ruleset has-default-UNIV* **by** *fast*
from *assms*(*1*) *collect-allow-compl-correct*[**where** *P*=*UNIV*] **have** *collect-allow-compl* γ *rs* {} = − *collect-allow* γ *rs UNIV* **by** *fastforce*
moreover with *collect-allow-sound-complete assms*(*1*) **have** ... = − {*p*. *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow*} **by** *fast*
ultimately show *?thesis* **using** *univ* **by** *fastforce*
qed

with *packet-set-to-set* ?γ (*packet-set-constrain ?a ?m ?P*) = {$p \in$ *packet-set-to-set* ?γ *?P*. *matches* ?γ *?m ?a p*} and *packet-set-to-set* ?γ (*packet-set-constrain-not ?a ?m ?P*) = {$p \in$ *packet-set-to-set* ?γ *?P*. ¬ *matches* ?γ *?m ?a p*}, it should be possible to build an executable version of the algorithm above.

22.4 The set of all accepted packets – Executable Implementation

fun *collect-allow-impl-v1* :: *'a rule list* ⇒ *'a packet-set* ⇒ *'a packet-set* **where**
collect-allow-impl-v1 [] *P* = *packet-set-Empty* |
collect-allow-impl-v1 ((*Rule m Accept*)#*rs*) *P* = *packet-set-union* (*packet-set-constrain Accept m P*) (*collect-allow-impl-v1 rs* (*packet-set-constrain-not Accept m P*)) |
collect-allow-impl-v1 ((*Rule m Drop*)#*rs*) *P* = (*collect-allow-impl-v1 rs* (*packet-set-constrain-not Drop m P*))

lemma *collect-allow-impl-v1*: *simple-ruleset rs* ⟹ *packet-set-to-set* γ (*collect-allow-impl-v1 rs P*) = *collect-allow* γ *rs* (*packet-set-to-set* γ *P*)
apply(*induction* γ *rs* (*packet-set-to-set* γ *P*) *arbitrary*: *P rule*: *collect-allow.induct*)
apply(*simp-all add*: *packet-set-union-correct packet-set-constrain-correct packet-set-constrain-not-correct packet-set-Empty simple-ruleset-def*)
done

fun *collect-allow-impl-v2* :: *'a rule list* ⇒ *'a packet-set* ⇒ *'a packet-set* **where**
collect-allow-impl-v2 [] *P* = *packet-set-Empty* |
collect-allow-impl-v2 ((*Rule m Accept*)#*rs*) *P* = *packet-set-opt* (*packet-set-union*

(*packet-set-opt* (*packet-set-constrain Accept m P*)) (*packet-set-opt* (*collect-allow-impl-v2 rs* (*packet-set-opt* (*packet-set-constrain-not Accept m* (*packet-set-opt P*)))))) |
collect-allow-impl-v2 ((*Rule m Drop*)#*rs*) *P* = (*collect-allow-impl-v2 rs* (*packet-set-opt* (*packet-set-constrain-not Drop m* (*packet-set-opt P*))))

lemma *collect-allow-impl-v2*: *simple-ruleset rs* ⟹ *packet-set-to-set* γ (*collect-allow-impl-v2 rs P*) = *packet-set-to-set* γ (*collect-allow-impl-v1 rs P*)
apply(*induction rs P arbitrary*: *P rule*: *collect-allow-impl-v1.induct*)

apply(*simp-all add*: *simple-ruleset-def packet-set-union-correct packet-set-opt-correct packet-set-constrain-not-correct collect-allow-impl-v1*)
done

executable!

export-code *collect-allow-impl-v2* **in** *SML*

theorem *collect-allow-impl-v1-sound-complete*: *simple-ruleset rs* $\Longrightarrow$
packet-set-to-set γ (*collect-allow-impl-v1 rs packet-set-UNIV*) = $\{p.$ *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow*$\}$
apply(*simp add*: *collect-allow-impl-v1 packet-set-UNIV*)
using *collect-allow-sound-complete* **by** *fast*

corollary *collect-allow-impl-v2-sound-complete*: *simple-ruleset rs* $\Longrightarrow$
packet-set-to-set γ (*collect-allow-impl-v2 rs packet-set-UNIV*) = $\{p.$ *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow*$\}$
using *collect-allow-impl-v1-sound-complete collect-allow-impl-v2* **by** *fast*

instead of the expensive invert and intersect operations, we try to build the algorithm primarily by union

lemma $(UNIV - A) \cap (UNIV - B) = UNIV - (A \cup B)$ **by** *blast*
lemma $A \cap (- P) = UNIV - (-A \cup P)$ **by** *blast*
lemma $UNIV - ((- P) \cap A) = P \cup - A$ **by** *blast*
lemma $((- P) \cap A) = UNIV - (P \cup - A)$ **by** *blast*

lemma $UNIV - ((P \cup - A) \cap X) = UNIV - ((P \cap X) \cup (- A \cap X))$ **by** *blast*
lemma $UNIV - ((P \cap X) \cup (- A \cap X)) = (- P \cup -X) \cap (A \cup - X)$ **by** *blast*
lemma $(- P \cup -X) \cap (A \cup -X) = (- P \cap A) \cup - X$ **by** *blast*

lemma $(((- P) \cap A) \cup X) = UNIV - ((P \cup - A) \cap - X)$ **by** *blast*

lemma *set-helper1*:
$(- P \cap - \{p.\ matches\ \gamma\ m\ a\ p\}) = \{p.\ p \notin P \wedge \neg\ matches\ \gamma\ m\ a\ p\}$
$- \{p \in - P.\ matches\ \gamma\ m\ a\ p\} = (P \cup - \{p.\ matches\ \gamma\ m\ a\ p\})$
$- \{p.\ matches\ \gamma\ m\ a\ p\} = \{p.\ \neg\ matches\ \gamma\ m\ a\ p\}$
by *blast+*

fun *collect-allow-compl-impl* :: *'a rule list* $\Rightarrow$ *'a packet-set* $\Rightarrow$ *'a packet-set* **where**
collect-allow-compl-impl [] *P* = *packet-set-UNIV* |
collect-allow-compl-impl ((*Rule m Accept*)#*rs*) *P* = *packet-set-intersect* (*packet-set-union P* (*packet-set-not* (*to-packet-set Accept m*))) (*collect-allow-compl-impl rs* (*packet-set-opt* (*packet-set-union P* (*to-packet-set Accept m*)))) |
collect-allow-compl-impl ((*Rule m Drop*)#*rs*) *P* = (*collect-allow-compl-impl rs* (*packet-set-opt* (*packet-set-union P* (*to-packet-set Drop m*))))

lemma *collect-allow-compl-impl*: *simple-ruleset rs* $\Longrightarrow$

packet-set-to-set γ (*collect-allow-compl-impl rs P*) = − *collect-allow* γ *rs* (− *packet-set-to-set* γ *P*)
apply(*simp add*: *collect-allow-compl-correct*[*symmetric*])
apply(*induction rs P arbitrary*: *P rule*: *collect-allow-impl-v1.induct*)
apply(*simp-all add*: *simple-ruleset-def packet-set-union-correct packet-set-opt-correct packet-set-intersect-intersect packet-set-not-correct*
to-packet-set-set set-helper1 packet-set-UNIV)
done

take *UNIV* setminus the intersect over the result and get the set of allowed packets

fun *collect-allow-compl-impl-tailrec* :: *'a rule list* ⇒ *'a packet-set* ⇒ *'a packet-set list* ⇒ *'a packet-set list* **where**
collect-allow-compl-impl-tailrec [] *P PAs* = *PAs* |
collect-allow-compl-impl-tailrec ((*Rule m Accept*)#*rs*) *P PAs* =
collect-allow-compl-impl-tailrec rs (*packet-set-opt* (*packet-set-union P* (*to-packet-set Accept m*))) ((*packet-set-union P* (*packet-set-not* (*to-packet-set Accept m*)))# *PAs*) |
collect-allow-compl-impl-tailrec ((*Rule m Drop*)#*rs*) *P PAs* = *collect-allow-compl-impl-tailrec rs* (*packet-set-opt* (*packet-set-union P* (*to-packet-set Drop m*))) *PAs*

lemma *collect-allow-compl-impl-tailrec-helper*: *simple-ruleset rs* ⟹
(*packet-set-to-set* γ (*collect-allow-compl-impl rs P*)) ∩ (⋂ *set* (*map* (*packet-set-to-set* γ) *PAs*)) =
(⋂ *set* (*map* (*packet-set-to-set* γ) (*collect-allow-compl-impl-tailrec rs P PAs*)))
proof(*induction rs P arbitrary*: *PAs P rule*: *collect-allow-compl-impl.induct*)
case (*2 m rs*)
from *2* **have** *IH*: (⋀*P PAs*. *packet-set-to-set* γ (*collect-allow-compl-impl rs P*) ∩ (⋂*x*∈*set PAs*. *packet-set-to-set* γ *x*) =
(⋂*x*∈*set* (*collect-allow-compl-impl-tailrec rs P PAs*). *packet-set-to-set* γ *x*))
by(*simp add*: *simple-ruleset-def*)
from *IH*[**where** *P*=(*packet-set-opt* (*packet-set-union P* (*to-packet-set Accept m*))) **and** *PAs*=(*packet-set-union P* (*packet-set-not* (*to-packet-set Accept m*)) # *PAs*)] **have**
(*packet-set-to-set* γ *P* ∪ {*p*. ¬ *matches* γ *m Accept p*}) ∩
packet-set-to-set γ (*collect-allow-compl-impl rs* (*packet-set-opt* (*packet-set-union P* (*to-packet-set Accept m*)))) ∩
(⋂*x*∈*set PAs*. *packet-set-to-set* γ *x*) =
(⋂*x*∈*set*
(*collect-allow-compl-impl-tailrec rs* (*packet-set-opt* (*packet-set-union P* (*to-packet-set Accept m*))) (*packet-set-union P* (*packet-set-not* (*to-packet-set Accept m*)) # *PAs*)).
packet-set-to-set γ *x*)
apply(*simp add*: *packet-set-union-correct packet-set-not-correct to-packet-set-set*)
by *blast*
thus *?case*
by(*simp add*: *packet-set-union-correct packet-set-opt-correct packet-set-intersect-intersect packet-set-not-correct*

to-packet-set-set set-helper1 packet-set-constrain-not-correct)
qed(*simp-all add*: *simple-ruleset-def packet-set-union-correct packet-set-opt-correct packet-set-intersect-intersect packet-set-not-correct*
to-packet-set-set set-helper1 packet-set-constrain-not-correct packet-set-UNIV packet-set-Empty-def)

lemma *collect-allow-compl-impl-tailrec-correct*: *simple-ruleset rs* $\Longrightarrow$
(*packet-set-to-set* γ (*collect-allow-compl-impl rs P*)) = ($\bigcap$ $x\in$*set* (*collect-allow-compl-impl-tailrec rs P* [])). *packet-set-to-set* γ x)
using *collect-allow-compl-impl-tailrec-helper*[**where** *PAs*=[], *simplified*]
by *metis*

definition *allow-set-not-inter* :: *'a rule list* $\Rightarrow$ *'a packet-set list* **where**
allow-set-not-inter rs $\equiv$ *collect-allow-compl-impl-tailrec rs packet-set-Empty* []

Intersecting over the result of *allow-set-not-inter* and inverting is the list of all allowed packets

lemma *allow-set-not-inter*: *simple-ruleset rs* $\Longrightarrow$
$-$ ($\bigcap$ $x\in$*set* (*allow-set-not-inter rs*). *packet-set-to-set* γ x) = {*p*. *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalAllow*}
unfolding *allow-set-not-inter-def*
apply(*simp add*: *collect-allow-compl-impl-tailrec-correct*[*symmetric*])
apply(*simp add*:*collect-allow-compl-impl*)
apply(*simp add*: *packet-set-Empty*)
using *collect-allow-sound-complete* **by** *fast*

this gives the set of denied packets

lemma *simple-ruleset rs* $\Longrightarrow$ *has-default rs* $\Longrightarrow$
($\bigcap$ $x\in$*set* (*allow-set-not-inter rs*). *packet-set-to-set* γ x) = {*p*. *approximating-bigstep-fun* γ *p rs Undecided* = *Decision FinalDeny*}
apply(*frule simple-imp-good-ruleset*)
apply(*drule*(*1*) *has-default-UNIV*[**where** γ=γ])
apply(*drule allow-set-not-inter*[**where** γ=γ])

by *force*

lemma *UNIV* $-$ (($P \cup - A$) $\cap$ X) = $-$ (($-$($-$ $P \cap A$)) $\cap$ X) **by** *blast*

end
theory *Matching-Embeddings*
imports *Semantics-Ternary/Matching-Ternary Matching Semantics-Ternary/Unknown-Match-Tacs*
begin

23 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 $\exists\, m\ \beta\ \alpha\ a.$ *Matching-Ternary.matches* $(\beta, \alpha)\ m\ a\ p \neq$ *Semantics.matches* $(\lambda\ atm\ p.\ case\ \beta\ atm\ p\ of\ TernaryTrue \Rightarrow True \mid TernaryFalse \Rightarrow False \mid TernaryUnknown \Rightarrow \alpha\ a\ p)\ m\ p$
apply(*rule-tac x=MatchNot* (*Match X*) **in** *exI*) — any X
apply (*simp split*: *ternaryvalue.split ternaryvalue.split-asm add*: *matches-case-ternaryvalue-tuple bunch-of-lemmata-about-matches*)
by *fast*

the *the* in the next definition is always defined

lemma $\forall\, m \in \{m.\ approx\ m\ p \neq TernaryUnknown\}.$ *ternary-to-bool* $(approx\ m\ p) \neq None$
by(*simp add*: *ternary-to-bool-None*)

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 $\equiv \forall\, p\ m.\ approx\ m\ p \neq TernaryUnknown \longrightarrow exact\ m\ p = the$ (*ternary-to-bool* $(approx\ m\ p))$

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* $\longleftrightarrow (\forall\, p\ m.\ exact\ m\ p = the$ (*ternary-to-bool* $(approx\ m\ p)) \vee approx\ m\ p = TernaryUnknown)$
unfolding *matcher-agree-on-exact-matches-def* **by** *blast*

lemma *eval-ternary-Not-TrueD*: *eval-ternary-Not* $m = TernaryTrue \Longrightarrow m = TernaryFalse$
by (*metis eval-ternary-Not.simps*(*1*) *eval-ternary-idempotence-Not*)

lemma *matches-comply-exact*: *ternary-ternary-eval* (*map-match-tac* $\beta\ p\ m) \neq TernaryUnknown \Longrightarrow$
matcher-agree-on-exact-matches $\gamma\ \beta \Longrightarrow$
Semantics.matches $\gamma\ m\ p =$ *Matching-Ternary.matches* $(\beta, \alpha)\ m\ a\ p$
proof(*unfold matches-case-ternaryvalue-tuple*,*induction m*)
case *Match* **thus** *?case*
by(*simp split*: *ternaryvalue.split add*: *matcher-agree-on-exact-matches-def*)
next

case (*MatchNot m*) **thus** *?case*
apply(*simp split*: *ternaryvalue.split add*: *matcher-agree-on-exact-matches-def*)
apply(*case-tac ternary-ternary-eval* (*map-match-tac* β *p m*))
by(*simp-all*)
next
case (*MatchAnd m1 m2*)
thus *?case*
apply(*simp split*: *ternaryvalue.split-asm ternaryvalue.split*)
apply(*case-tac ternary-ternary-eval* (*map-match-tac* β *p m1*))
apply(*case-tac* [!] *ternary-ternary-eval* (*map-match-tac* β *p m2*))
by(*simp-all*)
next
case *MatchAny* **thus** *?case* **by** *simp*
qed

lemma *in-doubt-allow-allows-Accept*: $a = Accept \Longrightarrow$ *matcher-agree-on-exact-matches* $\gamma\ \beta \Longrightarrow$
Semantics.matches γ *m p* $\Longrightarrow$ *Matching-Ternary.matches* (β, *in-doubt-allow*) *m a p*
apply(*case-tac ternary-ternary-eval* (*map-match-tac* β *p m*) $\neq$ *TernaryUnknown*)
using *matches-comply-exact* **apply** *fast*
apply(*simp add*: *matches-case-ternaryvalue-tuple*)
done

lemma *not-exact-match-in-doubt-allow-approx-match*: *matcher-agree-on-exact-matches* $\gamma\ \beta \Longrightarrow a = Accept \vee a = Reject \vee a = Drop \Longrightarrow$
$\neg$ *Semantics.matches* γ *m p* $\Longrightarrow$
($a = Accept \wedge$ *Matching-Ternary.matches* (β, *in-doubt-allow*) *m a p*) $\vee\ \neg$ *Matching-Ternary.matches* (β, *in-doubt-allow*) *m a p*
apply(*case-tac ternary-ternary-eval* (*map-match-tac* β *p m*) $\neq$ *TernaryUnknown*)
apply(*drule*(*1*) *matches-comply-exact*[**where** α=*in-doubt-allow* **and** *a*=*a*])
apply(*rule disjI2*)
apply *fast*
apply(*simp*)
apply(*clarify*)
apply(*simp add*: *matches-case-ternaryvalue-tuple*)
apply(*cases a*)
apply(*simp-all*)
done

lemma *in-doubt-deny-denies-DropReject*: $a = Drop \vee a = Reject \Longrightarrow$ *matcher-agree-on-exact-matches* $\gamma\ \beta \Longrightarrow$
Semantics.matches γ *m p* $\Longrightarrow$ *Matching-Ternary.matches* (β, *in-doubt-deny*)

m a p
apply(*case-tac ternary-ternary-eval* (*map-match-tac* β *p m*) $\neq$ *TernaryUnknown*)
using *matches-comply-exact* **apply** *fast*
apply(*simp*)
apply(*auto simp add*: *matches-case-ternaryvalue-tuple*)
done

lemma *not-exact-match-in-doubt-deny-approx-match*: *matcher-agree-on-exact-matches* γ $\beta \Longrightarrow a = Accept \vee a = Reject \vee a = Drop \Longrightarrow$
$\neg$ *Semantics.matches* γ *m p* $\Longrightarrow$
(($a = Drop \vee a = Reject$) $\wedge$ *Matching-Ternary.matches* (β, *in-doubt-deny*) *m a p*) $\vee$ $\neg$ *Matching-Ternary.matches* (β, *in-doubt-deny*) *m a p*
apply(*case-tac ternary-ternary-eval* (*map-match-tac* β *p m*) $\neq$ *TernaryUnknown*)
apply(*drule*(*1*) *matches-comply-exact*[**where** α=*in-doubt-deny* **and** *a*=*a*])
apply(*rule disjI2*)
apply *fast*
apply(*simp*)
apply(*clarify*)
apply(*simp add*: *matches-case-ternaryvalue-tuple*)
apply(*cases a*)
apply(*simp-all*)
done

The ternary primitive matcher can return exactly the result of the Boolean primitive matcher

definition β_{magic} :: ('*a*, '*p*) *matcher* $\Rightarrow$ ('*a* $\Rightarrow$ '*p* $\Rightarrow$ *ternaryvalue*) **where**
β_{magic} $\gamma \equiv$ (λ *a p*. *if* γ *a p then TernaryTrue else TernaryFalse*)

lemma *matcher-agree-on-exact-matches* γ (β_{magic} γ)
by(*simp add*: *matcher-agree-on-exact-matches-def* β_{magic}-*def*)

lemma β_{magic}-*not-Unknown*: *ternary-ternary-eval* (*map-match-tac* (β_{magic} γ) *p m*) $\neq$ *TernaryUnknown*
proof(*induction m*)
case *MatchNot* **thus** *?case* **using** *eval-ternary-Not-UnknownD* β_{magic}-*def*
by (*simp*) *blast*
case (*MatchAnd m1 m2*) **thus** *?case*
apply(*case-tac ternary-ternary-eval* (*map-match-tac* (β_{magic} γ) *p m1*))
apply(*case-tac* [!] *ternary-ternary-eval* (*map-match-tac* (β_{magic} γ) *p m2*))
by(*simp-all add*: β_{magic}-*def*)
qed (*simp-all add*: β_{magic}-*def*)

lemma β_{magic}-*matching*: *Matching-Ternary.matches* ((β_{magic} γ), α) *m a p* $\longleftrightarrow$ *Semantics.matches* γ *m p*
proof(*induction m*)
case *Match* **thus** *?case*
by(*simp add*: β_{magic}-*def matches-case-ternaryvalue-tuple*)
case *MatchNot* **thus** *?case*
by(*simp add*: *matches-case-ternaryvalue-tuple* β_{magic}-*not-Unknown split*: *ternary-*

value.split-asm)
qed (*simp-all add*: *matches-case-ternaryvalue-tuple split*: *ternaryvalue.split ternary-value.split-asm*)

end
theory *Semantics-Embeddings*
imports *Matching-Embeddings Semantics Semantics-Ternary/Semantics-Ternary*
begin

24 Semantics Embedding

24.1 Tactic *in-doubt-allow*

lemma *iptables-bigstep-undecided-to-undecided-in-doubt-allow-approx*: *matcher-agree-on-exact-matches* γ β $\Longrightarrow$
good-ruleset rs $\Longrightarrow$
$\Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Undecided \Longrightarrow$
$(\beta,\ in\text{-}doubt\text{-}allow),p\vdash \langle rs,\ Undecided\rangle \Rightarrow_\alpha Undecided \vee (\beta,\ in\text{-}doubt\text{-}allow),p\vdash \langle rs,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow$
apply(*rotate-tac 2*)
apply(*induction rs Undecided Undecided rule*: *iptables-bigstep-induct*)
apply(*simp-all*)
apply (*metis approximating-bigstep.skip*)
apply (*metis approximating-bigstep.empty approximating-bigstep.log approximating-bigstep.nomatch*)
apply(*case-tac a = Log*)
apply (*metis approximating-bigstep.log approximating-bigstep.nomatch*)
apply(*case-tac a = Empty*)
apply (*metis approximating-bigstep.empty approximating-bigstep.nomatch*)
apply(*drule-tac a=a* **in** *not-exact-match-in-doubt-allow-approx-match*)
apply(*simp-all*)
apply(*simp add*: *good-ruleset-alt*)
apply *fast*
apply (*metis approximating-bigstep.accept approximating-bigstep.nomatch*)
apply(*frule iptables-bigstep-to-undecided*)
apply(*simp*)
apply(*simp add*: *good-ruleset-append*)
apply (*metis* (*hide-lams*, *no-types*) *approximating-bigstep.decision Semantics-Ternary.seq'*)
apply(*simp add*: *good-ruleset-def*)
apply(*simp add*: *good-ruleset-def*)
done

lemma *FinalAllow-approximating-in-doubt-allow*: *matcher-agree-on-exact-matches* γ β $\Longrightarrow$
good-ruleset rs $\Longrightarrow$
$\Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Decision\ FinalAllow \Longrightarrow (\beta,\ in\text{-}doubt\text{-}allow),p\vdash \langle rs,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow$

apply(*rotate-tac 2*)
apply(*induction rs Undecided Decision FinalAllow rule*: *iptables-bigstep-induct*)
apply(*simp-all*)
apply (*metis approximating-bigstep.accept in-doubt-allow-allows-Accept*)
apply(*case-tac t*)
apply(*simp-all*)
prefer *2*
apply(*simp add*: *good-ruleset-append*)
apply (*metis approximating-bigstep.decision approximating-bigstep.seq Semantics.decisionD state.inject*)
apply(*thin-tac False* $\Longrightarrow$ - $\Longrightarrow$ -)
apply(*simp add*: *good-ruleset-append*, *clarify*)
apply(*drule*(*2*) *iptables-bigstep-undecided-to-undecided-in-doubt-allow-approx*)
apply(*erule disjE*)
apply (*metis approximating-bigstep.seq*)
apply (*metis approximating-bigstep.decision Semantics-Ternary.seq'*)
apply(*simp add*: *good-ruleset-alt*)
done

corollary *FinalAllows-subseteq-in-doubt-allow*: *matcher-agree-on-exact-matches* γ $\beta \Longrightarrow$ *good-ruleset rs* $\Longrightarrow$
$\{p.\ \Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Decision\ FinalAllow\} \subseteq \{p.\ (\beta,\ in\text{-}doubt\text{-}allow),p\vdash \langle rs,\ Undecided\rangle \Rightarrow_\alpha Decision\ FinalAllow\}$
using *FinalAllow-approximating-in-doubt-allow* **by** (*metis* (*lifting*, *full-types*) *Collect-mono*)

lemma *approximating-bigstep-undecided-to-undecided-in-doubt-allow-approx*: *matcher-agree-on-exact-matches* γ $\beta \Longrightarrow$
good-ruleset rs $\Longrightarrow$
$(\beta,\ in\text{-}doubt\text{-}allow),p\vdash \langle rs,\ Undecided\rangle \Rightarrow_\alpha Undecided \Longrightarrow \Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Undecided \lor \Gamma,\gamma,p\vdash \langle rs,\ Undecided\rangle \Rightarrow Decision\ FinalDeny$
apply(*rotate-tac 2*)
apply(*induction rs Undecided Undecided rule*: *approximating-bigstep-induct*)
apply(*simp-all*)
apply (*metis iptables-bigstep.skip*)
apply (*metis iptables-bigstep.empty iptables-bigstep.log iptables-bigstep.nomatch*)
apply(*simp split*: *ternaryvalue.split-asm add*: *matches-case-ternaryvalue-tuple*)
apply (*metis in-doubt-allow-allows-Accept iptables-bigstep.nomatch matches-casesE ternaryvalue.distinct*(*1*) *ternaryvalue.distinct*(*5*))
apply(*case-tac a*)
apply(*simp-all*)
apply (*metis iptables-bigstep.drop iptables-bigstep.nomatch*)
apply (*metis iptables-bigstep.log iptables-bigstep.nomatch*)
apply (*metis iptables-bigstep.nomatch iptables-bigstep.reject*)
apply(*simp add*: *good-ruleset-alt*)
apply(*simp add*: *good-ruleset-alt*)
apply (*metis iptables-bigstep.empty iptables-bigstep.nomatch*)

apply(*simp add*: *good-ruleset-alt*)
apply(*simp add*: *good-ruleset-append*,*clarify*)
by (*metis approximating-bigstep-to-undecided iptables-bigstep.decision iptables-bigstep.seq*)

lemma *FinalDeny-approximating-in-doubt-allow*: *matcher-agree-on-exact-matches* γ β $\Longrightarrow$
good-ruleset rs $\Longrightarrow$
(β, *in-doubt-allow*),$p\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalDeny* $\Longrightarrow$ Γ,γ,$p\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalDeny*
apply(*rotate-tac 2*)
apply(*induction rs Undecided Decision FinalDeny rule*: *approximating-bigstep-induct*)
apply(*simp-all*)
apply (*metis action.distinct*(*1*) *action.distinct*(*5*) *deny not-exact-match-in-doubt-allow-approx-match*)

apply(*simp add*: *good-ruleset-append*, *clarify*)
apply(*case-tac t*)
apply(*simp*)
apply(*drule*(*2*) *approximating-bigstep-undecided-to-undecided-in-doubt-allow-approx*[**where** Γ=Γ])
apply(*erule disjE*)
apply (*metis iptables-bigstep.seq*)
apply (*metis iptables-bigstep.decision iptables-bigstep.seq*)
by (*metis Decision-approximating-bigstep-fun approximating-semantics-imp-fun iptables-bigstep.decision iptables-bigstep.seq*)

corollary *FinalDenys-subseteq-in-doubt-allow*: *matcher-agree-on-exact-matches* γ β $\Longrightarrow$ *good-ruleset rs* $\Longrightarrow$
{*p*. (β, *in-doubt-allow*),$p\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalDeny*} $\subseteq$ {*p*. Γ,γ,$p\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalDeny*}
using *FinalDeny-approximating-in-doubt-allow* **by** (*metis* (*lifting*, *full-types*) *Collect-mono*)

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* γ β $\Longrightarrow$ *good-ruleset rs* $\Longrightarrow$
approximating-bigstep-fun (β, *in-doubt-allow*) *p rs Undecided* = (*Decision FinalDeny*) $\Longrightarrow$ Γ,γ,$p\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalDeny*
apply(*frule*(*1*) *FinalDeny-approximating-in-doubt-allow*[**where** *p*=*p* **and** Γ=Γ])
apply(*rule approximating-fun-imp-semantics*)
apply (*metis good-imp-wf-ruleset*)
apply(*simp-all*)
done

24.2 Tactic *in-doubt-deny*

lemma *iptables-bigstep-undecided-to-undecided-in-doubt-deny-approx*: *matcher-agree-on-exact-matches* γ β $\Longrightarrow$
good-ruleset rs $\Longrightarrow$
Γ,γ,$p\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided* $\Longrightarrow$

(β, *in-doubt-deny*),$p\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Undecided* $\vee$ (β, *in-doubt-deny*),$p\vdash$
$\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalDeny*
apply(*rotate-tac 2*)
apply(*induction rs Undecided Undecided rule*: *iptables-bigstep-induct*)
apply(*simp-all*)
apply (*metis approximating-bigstep.skip*)
apply (*metis approximating-bigstep.empty approximating-bigstep.log approximating-bigstep.nomatch*)
apply(*case-tac a = Log*)
apply (*metis approximating-bigstep.log approximating-bigstep.nomatch*)
apply(*case-tac a = Empty*)
apply (*metis approximating-bigstep.empty approximating-bigstep.nomatch*)
apply(*drule-tac a=a* **in** *not-exact-match-in-doubt-deny-approx-match*)
apply(*simp-all*)
apply(*simp add*: *good-ruleset-alt*)
apply *fast*
apply (*metis approximating-bigstep.drop approximating-bigstep.nomatch approximating-bigstep.reject*)
apply(*frule iptables-bigstep-to-undecided*)
apply(*simp*)
apply(*simp add*: *good-ruleset-append*)
apply (*metis* (*hide-lams*, *no-types*) *approximating-bigstep.decision Semantics-Ternary.seq′*)
apply(*simp add*: *good-ruleset-def*)
apply(*simp add*: *good-ruleset-def*)
done

lemma *FinalDeny-approximating-in-doubt-deny*: *matcher-agree-on-exact-matches*
γ β $\Longrightarrow$
good-ruleset rs $\Longrightarrow$
Γ,γ,$p\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalDeny* $\Longrightarrow$ (β, *in-doubt-deny*),$p\vdash$ $\langle$*rs*,
Undecided$\rangle$ $\Rightarrow_\alpha$ *Decision FinalDeny*
apply(*rotate-tac 2*)
apply(*induction rs Undecided Decision FinalDeny rule*: *iptables-bigstep-induct*)
apply(*simp-all*)
apply (*metis approximating-bigstep.drop approximating-bigstep.reject in-doubt-deny-denies-DropReject*)
apply(*case-tac t*)
apply(*simp-all*)
prefer *2*
apply(*simp add*: *good-ruleset-append*)
apply(*thin-tac False* $\Longrightarrow$ -)
apply (*metis approximating-bigstep.decision approximating-bigstep.seq Semantics.decisionD state.inject*)
apply(*thin-tac False* $\Longrightarrow$ - $\Longrightarrow$ -)
apply(*simp add*: *good-ruleset-append*, *clarify*)

apply(*drule*(*2*) *iptables-bigstep-undecided-to-undecided-in-doubt-deny-approx*)
apply(*erule disjE*)
apply (*metis approximating-bigstep.seq*)
apply (*metis approximating-bigstep.decision Semantics-Ternary.seq′*)
apply(*simp add*: *good-ruleset-alt*)

done

lemma *approximating-bigstep-undecided-to-undecided-in-doubt-deny-approx*: *matcher-agree-on-exact-matches* γ $\beta \Longrightarrow$
good-ruleset rs $\Longrightarrow$
(β, *in-doubt-deny*),$p\vdash$ $\langle$*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 FinalAllow*
apply(*rotate-tac 2*)
apply(*induction rs Undecided Undecided rule*: *approximating-bigstep-induct*)
apply(*simp-all*)
apply (*metis iptables-bigstep.skip*)
apply (*metis iptables-bigstep.empty iptables-bigstep.log iptables-bigstep.nomatch*)
apply(*simp split*: *ternaryvalue.split-asm add*: *matches-case-ternaryvalue-tuple*)
apply (*metis in-doubt-allow-allows-Accept iptables-bigstep.nomatch matches-casesE ternaryvalue.distinct*(*1*) *ternaryvalue.distinct*(*5*))
apply(*case-tac a*)
apply(*simp-all*)
apply (*metis iptables-bigstep.accept iptables-bigstep.nomatch*)
apply (*metis iptables-bigstep.log iptables-bigstep.nomatch*)
apply(*simp add*: *good-ruleset-alt*)
apply(*simp add*: *good-ruleset-alt*)
apply (*metis iptables-bigstep.empty iptables-bigstep.nomatch*)
apply(*simp add*: *good-ruleset-alt*)
apply(*simp add*: *good-ruleset-append*,*clarify*)
by (*metis approximating-bigstep-to-undecided iptables-bigstep.decision iptables-bigstep.seq*)

lemma *FinalAllow-approximating-in-doubt-deny*: *matcher-agree-on-exact-matches* γ $\beta \Longrightarrow$
good-ruleset rs $\Longrightarrow$
(β, *in-doubt-deny*),$p\vdash$ $\langle$*rs*, *Undecided*$\rangle \Rightarrow_\alpha$ *Decision FinalAllow* $\Longrightarrow \Gamma,\gamma,p\vdash$ $\langle$*rs*, *Undecided*$\rangle \Rightarrow$ *Decision FinalAllow*
apply(*rotate-tac 2*)
apply(*induction rs Undecided Decision FinalAllow rule*: *approximating-bigstep-induct*)
apply(*simp-all*)
apply (*metis action.distinct*(*1*) *action.distinct*(*5*) *iptables-bigstep.accept not-exact-match-in-doubt-deny-appro*
apply(*simp add*: *good-ruleset-append*, *clarify*)
apply(*case-tac t*)
apply(*simp*)
apply(*drule*(*2*) *approximating-bigstep-undecided-to-undecided-in-doubt-deny-approx*[**where** Γ=Γ])
apply(*erule disjE*)
apply (*metis iptables-bigstep.seq*)
apply (*metis iptables-bigstep.decision iptables-bigstep.seq*)
by (*metis Decision-approximating-bigstep-fun approximating-semantics-imp-fun iptables-bigstep.decision iptables-bigstep.seq*)

corollary *FinalAllows-subseteq-in-doubt-deny*: *matcher-agree-on-exact-matches* γ $\beta \Longrightarrow$ *good-ruleset rs* $\Longrightarrow$
$\{p.\ (\beta,\ \textit{in-doubt-deny}),p\vdash \langle rs,\ \textit{Undecided}\rangle \Rightarrow_\alpha \textit{Decision FinalAllow}\} \subseteq \{p.\ \Gamma,\gamma,p\vdash \langle rs,\ \textit{Undecided}\rangle \Rightarrow \textit{Decision FinalAllow}\}$
using *FinalAllow-approximating-in-doubt-deny* **by** (*metis* (*lifting*, *full-types*) *Collect-mono*)

24.3 Approximating Closures

theorem *FinalAllowClosure*:
assumes *matcher-agree-on-exact-matches* γ β **and** *good-ruleset rs*
shows $\{p.\ (\beta,\ \textit{in-doubt-deny}),p\vdash \langle rs,\ \textit{Undecided}\rangle \Rightarrow_\alpha \textit{Decision FinalAllow}\} \subseteq \{p.\ \Gamma,\gamma,p\vdash \langle rs,\ \textit{Undecided}\rangle \Rightarrow \textit{Decision FinalAllow}\}$
and $\{p.\ \Gamma,\gamma,p\vdash \langle rs,\ \textit{Undecided}\rangle \Rightarrow \textit{Decision FinalAllow}\} \subseteq \{p.\ (\beta,\ \textit{in-doubt-allow}),p\vdash \langle rs,\ \textit{Undecided}\rangle \Rightarrow_\alpha \textit{Decision FinalAllow}\}$
apply (*metis FinalAllows-subseteq-in-doubt-deny assms*)
by (*metis FinalAllows-subseteq-in-doubt-allow assms*)

theorem *FinalDenyClosure*:
assumes *matcher-agree-on-exact-matches* γ β **and** *good-ruleset rs*
shows $\{p.\ (\beta,\ \textit{in-doubt-allow}),p\vdash \langle rs,\ \textit{Undecided}\rangle \Rightarrow_\alpha \textit{Decision FinalDeny}\} \subseteq \{p.\ \Gamma,\gamma,p\vdash \langle rs,\ \textit{Undecided}\rangle \Rightarrow \textit{Decision FinalDeny}\}$
and $\{p.\ \Gamma,\gamma,p\vdash \langle rs,\ \textit{Undecided}\rangle \Rightarrow \textit{Decision FinalDeny}\} \subseteq \{p.\ (\beta,\ \textit{in-doubt-deny}),p\vdash \langle rs,\ \textit{Undecided}\rangle \Rightarrow_\alpha \textit{Decision FinalDeny}\}$
apply (*metis FinalDenys-subseteq-in-doubt-allow assms*)
by (*metis FinalDeny-approximating-in-doubt-deny assms mem-Collect-eq subsetI*)

24.4 Exact Embedding

thm *matcher-agree-on-exact-matches-def*[*of* γ β]
lemma *LukassLemma*:
matcher-agree-on-exact-matches γ $\beta \Longrightarrow$
$(\forall\ r \in \textit{set rs}.\ \textit{ternary-ternary-eval}\ (\textit{map-match-tac}\ \beta\ p\ (\textit{get-match}\ r)) \neq \textit{TernaryUnknown}) \Longrightarrow$
good-ruleset rs $\Longrightarrow$
$(\beta,\alpha),p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t \Longrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
apply(*simp add*: *matcher-agree-on-exact-matches-def*)
apply(*rotate-tac 3*)
apply(*induction rs s t rule*: *approximating-bigstep-induct*)
apply(*auto intro*: *approximating-bigstep.intros iptables-bigstep.intros dest*: *iptables-bigstepD*)
apply (*metis iptables-bigstep.accept matcher-agree-on-exact-matches-def matches-comply-exact*)
apply (*metis deny matcher-agree-on-exact-matches-def matches-comply-exact*)
apply (*metis iptables-bigstep.reject matcher-agree-on-exact-matches-def matches-comply-exact*)
apply (*metis iptables-bigstep.nomatch matcher-agree-on-exact-matches-def matches-comply-exact*)
by (*metis good-ruleset-append iptables-bigstep.seq*)

For rulesets without *Call*s, the approximating ternary semantics can perfectly simulate the Boolean semantics.

theorem β_{magic}*-approximating-bigstep-iff-iptables-bigstep*:

assumes $\forall r \in set\ rs.\ \forall c.\ get\text{-}action\ r \neq Call\ c$
shows $((\beta_{magic}\ \gamma),\alpha),p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t \longleftrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
apply(*rule iffI*)
apply(*induction rs s t rule*: *approximating-bigstep-induct*)
apply(*auto intro*: *iptables-bigstep.intros simp*: β_{magic}*-matching*)[*7*]
apply(*insert assms*)
apply(*induction rs s t rule*: *iptables-bigstep-induct*)
apply(*auto intro*: *approximating-bigstep.intros simp*: β_{magic}*-matching*)
done

corollary β_{magic}*-approximating-bigstep-fun-iff-iptables-bigstep*:
assumes *good-ruleset rs*
shows $approximating\text{-}bigstep\text{-}fun\ (\beta_{magic}\ \gamma,\alpha)\ p\ rs\ s = t \longleftrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
apply(*subst approximating-semantics-iff-fun-good-ruleset*[*symmetric*])
using *assms* **apply** *simp*
apply(*subst* β_{magic}*-approximating-bigstep-iff-iptables-bigstep*[**where** Γ=Γ])
using *assms* **apply** (*simp add*: *good-ruleset-def*)
by *simp*

end
theory *Iptables-Semantics*
imports *Semantics-Embeddings Semantics-Ternary/Fixed-Action*
begin

25 Normalizing Rulesets in the Boolean Big Step Semantics

corollary *normalize-rules-dnf-correct-BooleanSemantics*:
assumes *good-ruleset rs*
shows $\Gamma,\gamma,p\vdash \langle normalize\text{-}rules\text{-}dnf\ rs,\ s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
proof −
from *assms* **have** *assm'*: *good-ruleset* (*normalize-rules-dnf rs*) **by** (*metis good-ruleset-normalize-rules-dnf*)

from *normalize-rules-dnf-correct assms good-imp-wf-ruleset* **have**
$\forall \beta\ \alpha.\ approximating\text{-}bigstep\text{-}fun\ (\beta,\alpha)\ p\ (normalize\text{-}rules\text{-}dnf\ rs)\ s = approximating\text{-}bigstep\text{-}fun\ (\beta,\alpha)\ p\ rs\ s$ **by** *fast*
hence
$\forall \alpha.\ approximating\text{-}bigstep\text{-}fun\ (\beta_{magic}\ \gamma,\alpha)\ p\ (normalize\text{-}rules\text{-}dnf\ rs)\ s = approximating\text{-}bigstep\text{-}fun\ (\beta_{magic}\ \gamma,\alpha)\ p\ rs\ s$ **by** *fast*
with β_{magic}*-approximating-bigstep-fun-iff-iptables-bigstep assms assm'* **show** *?thesis*
by *metis*
qed

end
theory *Optimizing*
imports *Semantics-Ternary Packet-Set-Impl*

begin

26 Optimizing

26.1 Removing Shadowed Rules

Assumes: *simple-ruleset*

fun *rmshadow* :: $('a, 'p)$ *match-tac* $\Rightarrow$ *'a rule list* $\Rightarrow$ *'p set* $\Rightarrow$ *'a rule list* **where**
rmshadow - [] - = [] |
rmshadow γ ((*Rule m a*)#*rs*) *P* = (*if* ($\forall p \in P$. $\neg$ *matches* γ *m a p*)
then
rmshadow γ *rs P*
else
(*Rule m a*) # (*rmshadow* γ *rs* $\{p \in P.\ \neg$ *matches* γ *m a p*$\}$))

26.1.1 Soundness

lemma *rmshadow-sound*:
simple-ruleset rs $\Longrightarrow$ $p \in P$ $\Longrightarrow$ *approximating-bigstep-fun* γ *p* (*rmshadow* γ *rs P*) = *approximating-bigstep-fun* γ *p rs*
proof(*induction rs arbitrary*: *P*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
let *?fw=approximating-bigstep-fun* γ — firewall semantics
let *?rm=rmshadow* γ
let *?match=matches* γ (*get-match r*) (*get-action r*)
let *?set*=$\{p \in P.\ \neg$ *?match p*$\}$
from *Cons.IH Cons.prems* **have** *IH*: *?fw p* (*?rm rs P*) = *?fw p rs* **by** (*simp add*: *simple-ruleset-def*)
from *Cons.IH*[*of ?set*] *Cons.prems* **have** *IH'*: $p \in$ *?set* $\Longrightarrow$ *?fw p* (*?rm rs ?set*) = *?fw p rs* **by** (*simp add*: *simple-ruleset-def*)
from *Cons* **show** *?case*
proof(*cases* $\forall p \in P$. $\neg$ *?match p*) — the if-condition of rmshadow
case *True*
from *True* **have** *1*: *?rm* (*r*#*rs*) *P* = *?rm rs P*
apply(*cases r*)
apply(*rename-tac m a*)
apply(*clarify*)
apply(*simp*)
done
from *True Cons.prems* **have** *?fw p* (*r* # *rs*) = *?fw p rs*
apply(*cases r*)
apply(*rename-tac m a*)
apply(*simp add*: *fun-eq-iff*)
apply(*clarify*)
apply(*rename-tac s*)
apply(*case-tac s*)
apply(*simp*)

```
      apply(simp add: Decision-approximating-bigstep-fun)
      done
    from this IH have ?fw p (?rm rs P) = ?fw p (r#rs)  by simp
    thus ?fw p (?rm (r#rs) P) = ?fw p (r#rs)  using 1 by simp
  next
  case False — else
    have ?fw p (r # (?rm rs ?set)) = ?fw p (r # rs)
      proof(cases p ∈ ?set)
        case True
          from True IH' show ?fw p (r # (?rm rs ?set)) = ?fw p (r#rs)
            apply(cases r)
            apply(rename-tac m a)
            apply(simp add: fun-eq-iff)
            apply(clarify)
            apply(rename-tac s)
            apply(case-tac s)
             apply(simp)
            apply(simp add: Decision-approximating-bigstep-fun)
            done
        next
        case False
          from False Cons.prems have ?match p by simp
           from Cons.prems have get-action r = Accept ∨ get-action r = Drop
by(simp add: simple-ruleset-def)
          from this ‹?match p›show ?fw p (r # (?rm rs ?set)) = ?fw p (r#rs)
            apply(cases r)
            apply(rename-tac m a)
            apply(simp add: fun-eq-iff)
            apply(clarify)
            apply(rename-tac s)
            apply(case-tac s)
             apply(simp split:action.split)
             apply fast
            apply(simp add: Decision-approximating-bigstep-fun)
            done
      qed
    from False this show ?thesis
      apply(cases r)
      apply(rename-tac m a)
      apply(simp add: fun-eq-iff)
      apply(clarify)
      apply(rename-tac s)
      apply(case-tac s)
       apply(simp)
      apply(simp add: Decision-approximating-bigstep-fun)
      done
 qed
qed
```

fun *rmMatchFalse* :: *'a rule list* ⇒ *'a rule list* **where**
rmMatchFalse [] = [] |
rmMatchFalse ((*Rule* (*MatchNot MatchAny*) -)#*rs*) = *rmMatchFalse rs* |
rmMatchFalse (*r*#*rs*) = *r* # *rmMatchFalse rs*

lemma *rmMatchFalse-helper*: *m* ≠ *MatchNot MatchAny* ⟹ (*rmMatchFalse* (*Rule m a* # *rs*)) = *Rule m a* # (*rmMatchFalse rs*)
apply(*case-tac m*)
apply(*simp-all*)
apply(*rename-tac match-expr*)
apply(*case-tac match-expr*)
apply(*simp-all*)
done

lemma *rmMatchFalse-correct*: *approximating-bigstep-fun* γ *p* (*rmMatchFalse rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
apply(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)
apply(*simp*)
apply (*metis Decision-approximating-bigstep-fun*)
apply(*case-tac m* = *MatchNot MatchAny*)
apply(*simp*)
apply(*simp add*: *rmMatchFalse-helper*)
apply(*subgoal-tac m* ≠ *MatchNot MatchAny*)
apply(*drule-tac a*=*a* **and** *rs*=*rs* **in** *rmMatchFalse-helper*)
apply(*simp split*:*action.split*)
apply(*thin-tac a* = *x* ⟹ - **for** *x*)
apply(*thin-tac a* = *x* ⟹ - **for** *x*)
by (*metis bunch-of-lemmata-about-matches*(*3*))

end
theory *Primitive-Normalization*
imports ../*Semantics-Ternary*/*Negation-Type-Matching*
begin

27 Primitive Normalization

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 *normalized-n-primitive* :: ((*'a* ⇒ *bool*) × (*'a* ⇒ *'b*)) ⇒ (*'b* ⇒ *bool*) ⇒ *'a match-expr* ⇒ *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* ∧ *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*

The following function takes a tuple of functions ((*'a* ⇒ *bool*) × (*'a* ⇒ *'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* ⇒ *bool*) × (*'a* ⇒ *'b*)) ⇒ *'a match-expr* ⇒ (*'b negation-type list* × *'a match-expr*) **where**
primitive-extractor - *MatchAny* = ([], *MatchAny*) |
primitive-extractor (*disc*,*sel*) (*Match a*) = (*if disc a then* ([*Pos* (*sel a*)], *MatchAny*) *else* ([], *Match a*)) |
primitive-extractor (*disc*,*sel*) (*MatchNot* (*Match a*)) = (*if disc a then* ([*Neg* (*sel a*)], *MatchAny*) *else* ([], *MatchNot* (*Match a*))) |
primitive-extractor C (*MatchAnd ms1 ms2*) = (
    let (*a1'*, *ms1'*) = *primitive-extractor C ms1*;
        (*a2'*, *ms2'*) = *primitive-extractor C ms2*
    in (*a1'@a2'*, *MatchAnd ms1' ms2'*)) |
primitive-extractor - - = *undefined*

The first part returned by *primitive-extractor*, here *as*: A list of primitive match expressions. For example, let *m* = *MatchAnd* (*Src ip1*) (*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.

theorem *primitive-extractor-correct*: **assumes**
normalized-nnf-match m **and** *wf-disc-sel* (*disc*, *sel*) *C* **and** *primitive-extractor*

(*disc*, *sel*) *m* = (*as*, *ms*)
  shows *matches* γ (*alist-and* (*NegPos-map C as*)) *a p* $\wedge$ *matches* γ *ms a p* $\longleftrightarrow$ *matches* γ *m a p*
  and *normalized-nnf-match ms*
  and $\neg$ *has-disc disc ms*
  and $\forall$ *disc2*. $\neg$ *has-disc disc2 m* $\longrightarrow$ $\neg$ *has-disc disc2 ms*
  and $\forall$ *disc2 sel2*. *normalized-n-primitive* (*disc2*, *sel2*) *P m* $\longrightarrow$ *normalized-n-primitive* (*disc2*, *sel2*) *P ms*
proof −
  — better simplification rule
  from *assms* **have** *assm3'*: (*as*, *ms*) = *primitive-extractor* (*disc*, *sel*) *m* **by** *simp*
  with *assms*(*1*) *assms*(*2*) **show** *matches* γ (*alist-and* (*NegPos-map C as*)) *a p* $\wedge$ *matches* γ *ms a p* $\longleftrightarrow$ *matches* γ *m a p*
   proof(*induction* (*disc*, *sel*) *m arbitrary*: *as ms rule*: *primitive-extractor*.*induct*)
    case *4* **thus** *?case*
     apply(*simp split*: *split-if-asm split-split-asm add*: *NegPos-map-append*)
     apply(*auto simp add*: *alist-and-append bunch-of-lemmata-about-matches*)
     done
   qed(*simp-all add*: *bunch-of-lemmata-about-matches wf-disc-sel*.*simps split*: *split-if-asm*)

  from *assms*(*1*) *assm3'* **show** *normalized-nnf-match ms*
   proof(*induction* (*disc*, *sel*) *m arbitrary*: *as ms rule*: *primitive-extractor*.*induct*)
     case *2* **thus** *?case* **by**(*simp split*: *split-if-asm*)
     next
     case *3* **thus** *?case* **by**(*simp split*: *split-if-asm*)
     next
     case *4* **thus** *?case*
      apply(*clarify*)
      apply(*simp split*: *split-split-asm*)
      done
   qed(*simp-all*)

  from *assms*(*1*) *assm3'* **show** $\neg$ *has-disc disc ms*
   proof(*induction* (*disc*, *sel*) *m arbitrary*: *as ms rule*: *primitive-extractor*.*induct*)
   qed(*simp-all split*: *split-if-asm split-split-asm*)

  from *assms*(*1*) *assm3'* **show** $\forall$ *disc2*. $\neg$ *has-disc disc2 m* $\longrightarrow$ $\neg$ *has-disc disc2 ms*
   proof(*induction* (*disc*, *sel*) *m arbitrary*: *as ms rule*: *primitive-extractor*.*induct*)
     case *2* **thus** *?case* **by**(*simp split*: *split-if-asm*)
     next
     case *3* **thus** *?case* **by**(*simp split*: *split-if-asm*)
     next
     case *4* **thus** *?case* **by**(*simp split*: *split-split-asm*)
   qed(*simp-all*)

  from *assms*(*1*) *assm3'* **show** $\forall$ *disc2 sel2*. *normalized-n-primitive* (*disc2*, *sel2*) *P m* $\longrightarrow$ *normalized-n-primitive* (*disc2*, *sel2*) *P ms*

apply(*induction* (*disc*, *sel*) *m* *arbitrary*: *as ms rule*: *primitive-extractor.induct*)
apply(*simp*)
apply(*simp split*: *split-if-asm*)
apply(*simp split*: *split-if-asm*)
apply(*simp split*: *split-split-asm*)
apply(*simp-all*)
done
qed

lemma *primitive-extractor-matchesE*: *wf-disc-sel* (*disc*,*sel*) $C \Longrightarrow$ *normalized-nnf-match* $m \Longrightarrow$ *primitive-extractor* (*disc*, *sel*) m = (*as*, *ms*)
$\Longrightarrow$
(*normalized-nnf-match ms* $\Longrightarrow \neg$ *has-disc disc ms* $\Longrightarrow$ ($\forall$ *disc2*. $\neg$ *has-disc disc2* $m \longrightarrow \neg$ *has-disc disc2 ms*) $\Longrightarrow$ *matches-other* $\longleftrightarrow$ *matches* γ *ms a p*)
$\Longrightarrow$
matches γ (*alist-and* (*NegPos-map C as*)) *a p* $\wedge$ *matches-other* $\longleftrightarrow$ *matches* γ *m a p*
using *primitive-extractor-correct* **by** *metis*

lemma *primitive-extractor-matches-lastE*: *wf-disc-sel* (*disc*,*sel*) $C \Longrightarrow$ *normalized-nnf-match* $m \Longrightarrow$ *primitive-extractor* (*disc*, *sel*) m = (*as*, *ms*)
$\Longrightarrow$
(*normalized-nnf-match ms* $\Longrightarrow \neg$ *has-disc disc ms* $\Longrightarrow$ ($\forall$ *disc2*. $\neg$ *has-disc disc2* $m \longrightarrow \neg$ *has-disc disc2 ms*) $\Longrightarrow$ *matches* γ *ms a p*)
$\Longrightarrow$
matches γ (*alist-and* (*NegPos-map C as*)) *a p* $\longleftrightarrow$ *matches* γ *m a p*
using *primitive-extractor-correct* **by** *metis*

The lemmas ⟦*wf-disc-sel* (*?disc*, *?sel*) *?C*; *normalized-nnf-match ?m*; *primitive-extractor* (*?disc*, *?sel*) *?m* = (*?as*, *?ms*); ⟦*normalized-nnf-match ?ms*; $\neg$ *has-disc ?disc ?ms*; $\forall$ *disc2*. $\neg$ *has-disc disc2 ?m* $\longrightarrow \neg$ *has-disc disc2 ?ms*⟧ $\Longrightarrow$ *?matches-other* = *matches ?*γ *?ms ?a ?p*⟧ $\Longrightarrow$ (*matches ?*γ (*alist-and* (*NegPos-map ?C ?as*)) *?a ?p* $\wedge$ *?matches-other*) = *matches ?*γ *?m ?a ?p* and ⟦*wf-disc-sel* (*?disc*, *?sel*) *?C*; *normalized-nnf-match ?m*; *primitive-extractor* (*?disc*, *?sel*) *?m* = (*?as*, *?ms*); ⟦*normalized-nnf-match ?ms*; $\neg$ *has-disc ?disc ?ms*; $\forall$ *disc2*. $\neg$ *has-disc disc2 ?m* $\longrightarrow \neg$ *has-disc disc2 ?ms*⟧ $\Longrightarrow$ *matches ?*γ *?ms ?a ?p*⟧ $\Longrightarrow$ *matches ?*γ (*alist-and* (*NegPos-map ?C ?as*)) *?a ?p* = *matches ?*γ *?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*].

27.1 Normalizing and Optimizing Primitives

Normalize primitives by a function f with type *'b negation-type list* $\Rightarrow$ *'b 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 r
f [Neg 41, {20..50}, {30..50}] = [{30..40}, {42..50}]   input: conjunction lis
```

definition *normalize-primitive-extract* :: $(('a \Rightarrow bool) \times ('a \Rightarrow 'b)) \Rightarrow$
$('b \Rightarrow 'a) \Rightarrow$
$('b$ *negation-type list* $\Rightarrow 'b$ *list*$) \Rightarrow$
$'a$ *match-expr* $\Rightarrow$
$'a$ *match-expr list* **where**
normalize-primitive-extract (*disc-sel*) *C f m* = (*case primitive-extractor* (*disc-sel*) *m*
of (*spts*, *rst*) ⇒ *map* (λ*spt*. (*MatchAnd* (*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**
∀ *ml*. (*match-list* γ (*map* (*Match* ∘ *C*) (*f ml*)) *a p* ⟷ *matches* γ (*alist-and* (*NegPos-map C ml*)) *a p*)
shows *match-list* γ (*normalize-primitive-extract disc-sel C f m*) *a p* ⟷ *matches* γ *m a p*
proof −
obtain *as ms* **where** *pe*: *primitive-extractor disc-sel m* = (*as*, *ms*) **by** *fastforce*

from *pe primitive-extractor-correct*(*1*)[*OF assms*(*1*), **where** γ=γ **and** *a*=*a* **and** *p*=*p*] *assms*(*2*) **have**
matches γ *m a p* ⟷ *matches* γ (*alist-and* (*NegPos-map C as*)) *a p* ∧ *matches* γ *ms a p* **by**(*cases disc-sel*, *blast*)
also have ... ⟷ *match-list* γ (*map* (*Match* ∘ *C*) (*f as*)) *a p* ∧ *matches* γ *ms a p* **using** *assms*(*3*) **by** *simp*
also have ... ⟷ *match-list* γ (*map* (λ*spt*. *MatchAnd* (*Match* (*C spt*)) *ms*) (*f as*)) *a p*
by(*simp add*: *match-list-matches bunch-of-lemmata-about-matches*)
also have ... ⟷ *match-list* γ (*normalize-primitive-extract disc-sel C f m*) *a p*
by(*simp add*: *normalize-primitive-extract-def pe*)

finally show *?thesis* **by** *simp*
qed

thm *match-list-semantics*[*of* γ (*map* (*Match* $\circ$ *C*) (*f ml*)) *a p* [(*alist-and* (*NegPos-map* *C ml*))]]

corollary *normalize-primitive-extract-semantics*: **assumes** *normalized-nnf-match* *m* **and** *wf-disc-sel disc-sel C* **and**
$\forall$ *ml*. (*match-list* γ (*map* (*Match* $\circ$ *C*) (*f ml*)) *a p* $\longleftrightarrow$ *matches* γ (*alist-and* (*NegPos-map C ml*)) *a p*)
shows *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) (*normalize-primitive-extract* *disc-sel C f m*)) *s* =
approximating-bigstep-fun γ *p* [*Rule m a*] *s*
proof −
from *normalize-primitive-extract*[*OF assms*(*1*) *assms*(*2*) *assms*(*3*)] **have**
match-list γ (*normalize-primitive-extract disc-sel C f m*) *a p* = *matches* γ *m* *a p* **.**
also have ... $\longleftrightarrow$ *match-list* γ [*m*] *a p* **by** *simp*
finally show *?thesis* **using** *match-list-semantics*[*of* γ (*normalize-primitive-extract* *disc-sel C f m*) *a p* [*m*]] **by** *simp*
qed

lemma *normalize-primitive-extract-preserves-nnf-normalized*:
assumes *normalized-nnf-match m*
and *wf-disc-sel* (*disc*, *sel*) *C*
shows $\forall$ *mn* $\in$ *set* (*normalize-primitive-extract* (*disc*, *sel*) *C f m*). *normalized-nnf-match* *mn*
proof
fix *mn*
assume *assm2*: *mn* $\in$ *set* (*normalize-primitive-extract* (*disc*, *sel*) *C f m*)
obtain *as ms* **where** *as-ms*: *primitive-extractor* (*disc*, *sel*) *m* = (*as*, *ms*) **by** *fastforce*
from *as-ms primitive-extractor-correct*[*OF assms*(*1*) *assms*(*2*)] **have** *normalized-nnf-match* *ms* **by** *simp*
from *assm2 as-ms* **have** *normalize-primitive-extract-unfolded*: *mn* $\in$ ((λ*spt*. *MatchAnd* (*Match* (*C spt*)) *ms*) ‘ *set* (*f as*))
unfolding *normalize-primitive-extract-def* **by** *force*
with ‹*normalized-nnf-match ms*› **show** *normalized-nnf-match mn* **by** *fastforce*
qed

If something is normalized for disc2 and disc2 ≠ 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*. $\neg$ *disc2* (*C a*) — disc1 and disc2 match for different stuff. e.g. *Src-Ports* and *Dst-Ports*
shows $\forall$ *mn* $\in$ *set* (*normalize-primitive-extract* (*disc1*, *sel1*) *C f m*). *normalized-n-primitive* (*disc2*, *sel2*) *P mn*
proof
fix *mn*
assume *assm2*: *mn* $\in$ *set* (*normalize-primitive-extract* (*disc1*, *sel1*) *C f m*)
obtain *as ms* **where** *as-ms*: *primitive-extractor* (*disc1*, *sel1*) *m* = (*as*, *ms*) **by** *fastforce*
from *as-ms primitive-extractor-correct*[*OF assms*(*1*) *assms*(*3*)] **have**
$\neg$ *has-disc disc1 ms*
and *normalized-n-primitive* (*disc2*, *sel2*) *P ms*
apply −
apply(*fast*)
using *assms*(*2*) **by**(*fast*)
from *assm2 as-ms* **have** *normalize-primitive-extract-unfolded*: *mn* $\in$ ((λ*spt*. *MatchAnd* (*Match* (*C spt*)) *ms*) ' *set* (*f as*))
unfolding *normalize-primitive-extract-def* **by** *force*

from *normalize-primitive-extract-unfolded* **obtain** *Casms* **where** *Casms*: *mn* = (*MatchAnd* (*Match* (*C Casms*)) *ms*) **by** *blast*

from ‹*normalized-n-primitive* (*disc2*, *sel2*) *P ms*› *assms*(*4*) **have** *normalized-n-primitive* (*disc2*, *sel2*) *P* (*MatchAnd* (*Match* (*C Casms*)) *ms*)
by(*simp*)

with *Casms* **show** *normalized-n-primitive* (*disc2*, *sel2*) *P mn* **by** *blast*
qed

thm *wf-disc-sel.simps*
lemma *wf-disc-sel* (*disc*, *sel*) *C* $\Longrightarrow$ $\forall$ *x*. *disc* (*C x*) **quickcheck oops**
lemma *wf-disc-sel* (*disc*, *sel*) *C* $\Longrightarrow$ *disc* (*C x*) $\longrightarrow$ *sel* (*C x*) = *x*
by(*simp add*: *wf-disc-sel.simps*)

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'* $\in$ *set* (*f as*). *P a'*)
shows $\forall$ *m'* $\in$ *set* (*normalize-primitive-extract* (*disc*, *sel*) *C f m*). *normalized-n-primitive* (*disc*, *sel*) *P m'*
proof
fix *m'* **assume** *a*: *m'*$\in$*set* (*normalize-primitive-extract* (*disc*, *sel*) *C f m*)

have *nnf*: $\forall$ *m'* $\in$ *set* (*normalize-primitive-extract* (*disc*, *sel*) *C f m*). *normalized-nnf-match m'*
using *normalize-primitive-extract-preserves-nnf-normalized assms* **by** *blast*

with *a* **have** *normalized-m′*: *normalized-nnf-match m′* **by** *simp*

from *a* **obtain** *as ms* **where** *as-ms*: *primitive-extractor* (*disc*, *sel*) *m* = (*as*, *ms*)
unfolding *normalize-primitive-extract-def* **by** *fastforce*
with *a* **have** *prems*: *m′* ∈ *set* (*map* (*λspt*. *MatchAnd* (*Match* (*C spt*)) *ms*) (*f as*))
unfolding *normalize-primitive-extract-def* **by** *simp*

from *primitive-extractor-correct*(*2*)[*OF assms*(*1*) *assms*(*2*) *as-ms*] **have** *normalized-nnf-match ms* **.**

show *normalized-n-primitive* (*disc*, *sel*) *P m′*
proof(*cases f as* = [])
case *True* **thus** *normalized-n-primitive* (*disc*, *sel*) *P m′* **using** *prems* **by** *simp*
next
case *False*
with *prems* **obtain** *spt* **where** *m′* = *MatchAnd* (*Match* (*C spt*)) *ms* **and** *spt* ∈ *set* (*f as*) **by** *auto*

from *primitive-extractor-correct*(*3*)[*OF assms*(*1*) *assms*(*2*) *as-ms*] **have** ¬ *has-disc disc ms* **.**
with ‹*normalized-nnf-match ms*› **have** *normalized-n-primitive* (*disc*, *sel*) *P ms*
by(*induction* (*disc*, *sel*) *P ms rule*: *normalized-n-primitive.induct*) *simp-all*

from ‹*wf-disc-sel* (*disc*, *sel*) *C*› **have** (*sel* (*C spt*)) = *spt* **by**(*simp add*: *wf-disc-sel.simps*)
with *np* ‹*spt* ∈ *set* (*f as*)› **have** *P* (*sel* (*C spt*)) **by** *simp*

show *normalized-n-primitive* (*disc*, *sel*) *P m′*
apply(*simp add*: ‹*m′* = *MatchAnd* (*Match* (*C spt*)) *ms*›)
apply(*rule conjI*)
apply(*simp-all add*: ‹*normalized-n-primitive* (*disc*, *sel*) *P ms*›)
apply(*simp add*: ‹*P* (*sel* (*C spt*))›)
done
qed
qed

lemma *normalized-n-primitive disc-sel f m* ⟹ *normalized-nnf-match m*
apply(*induction disc-sel f m rule*: *normalized-n-primitive.induct*)
apply(*simp-all*)
oops

lemma *remove-unknowns-generic-not-has-disc*: ¬ *has-disc C m* ⟹ ¬ *has-disc C*

(*remove-unknowns-generic* γ *a m*)
  by(*induction* γ *a m rule*: *remove-unknowns-generic.induct*) (*simp-all*)

lemma *remove-unknowns-generic-normalized-n-primitive*: *normalized-n-primitive disc-sel f m* $\Longrightarrow$
    normalized-n-primitive disc-sel f (*remove-unknowns-generic* γ *a m*)
  proof(*induction* γ *a m rule*: *remove-unknowns-generic.induct*)
    case *6* **thus** *?case* **by**(*case-tac disc-sel*, *simp*)
  qed(*simp-all*)

end
theory *No-Spoof*
imports
        ../Semantics-Embeddings
        Common-Primitive-Matcher
        Primitive-Normalization
begin

28 No Spoofing

assumes: *simple-ruleset*

A mapping from an interface to its assigned ip addresses in CIDR notation

  type-synonym *ipassignment*=*iface* $\rightharpoonup$ (*ipv4addr* $\times$ *nat*) *list*

Sanity checking for an *ipassignment*.

warning if interface map has wildcards

  definition *ipassmt-sanity-haswildcards* :: *ipassignment* $\Rightarrow$ *bool* **where**
   ipassmt-sanity-haswildcards ipassmt $\equiv$ $\forall$ *iface* $\in$ *dom ipassmt*. $\neg$ *iface-is-wildcard iface*

Executable of the *ipassignment* is given as a list.

   lemma[*code-unfold*]: *ipassmt-sanity-haswildcards* (*map-of ipassmt*) $\longleftrightarrow$ ($\forall$ *iface* $\in$ *fst‘ set ipassmt*. $\neg$ *iface-is-wildcard iface*)
    by(*simp add*: *ipassmt-sanity-haswildcards-def Map.dom-map-of-conv-image-fst*)

   value(*code*) *ipassmt-sanity-haswildcards* (*map-of* [(*Iface ''eth1.1017''*, [(*ipv4addr-of-dotdecimal* (*131*,*159*,*14*,*240*), *28*)])])

  fun *collect-ifaces* :: *common-primitive rule list* $\Rightarrow$ *iface list* **where**
    collect-ifaces [] = [] |
    collect-ifaces ((*Rule m a*)#*rs*) = *filter* (λ*iface*. *iface* $\neq$ *ifaceAny*) (
                                    (*map* (λ*x*. *case x of Pos i* $\Rightarrow$ *i* | *Neg i* $\Rightarrow$ *i*) (*fst* (*primitive-extractor* (*is-Iiface*, *iiface-sel*) *m*))) @
                                    (*map* (λ*x*. *case x of Pos i* $\Rightarrow$ *i* | *Neg i* $\Rightarrow$ *i*) (*fst* (*primitive-extractor* (*is-Oiface*, *oiface-sel*) *m*))) @ *collect-ifaces rs*)

definition *ipassmt-sanity-defined* :: *common-primitive rule list* ⇒ *ipassignment* ⇒ *bool* **where**
ipassmt-sanity-defined rs ipassmt ≡ ∀ *iface* ∈ *set* (*collect-ifaces rs*). *iface* ∈ *dom ipassmt*

Executable code

lemma[*code*]: *ipassmt-sanity-defined rs ipassmt* ⟷ (∀ *iface* ∈ *set* (*collect-ifaces rs*). *ipassmt iface* ≠ *None*)
by(*simp add*: *ipassmt-sanity-defined-def Map.domIff*)

value(*code*) *ipassmt-sanity-defined* [*Rule* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *24*))) (*Match* (*IIface* (*Iface* ''*eth1.1017*'')))) *action.Accept*,
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*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*)])])

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.

definition *no-spoofing* :: *ipassignment* ⇒ *common-primitive rule list* ⇒ *bool* **where**
no-spoofing ipassmt rs ≡ ∀ *iface* ∈ *dom ipassmt*. ∀ *p*.
((*common-matcher*, *in-doubt-allow*),*p*(|*p-iiface*:=*iface-sel iface*|)⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow*) ⟶
p-src p ∈ (*ipv4cidr-union-set* (*set* (*the* (*ipassmt iface*))))

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 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*'' ↦ {(*ipv4addr-of-dotdecimal* (*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*'' ↦ {*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 examples below. Check Example 3 why it can be thought of as OUTGOING spoofing.

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-booloan-semantics-nospoofing*:
assumes *matcher-agree-on-exact-matches* γ *common-matcher* **and** *simple-ruleset rs* **and** *no-spoofing*: *no-spoofing ipassmt rs*
shows $\forall$ *iface* $\in$ *dom ipassmt*. $\forall$ *p*. (Γ,γ,*p*(|*p-iiface*:=*iface-sel iface*|)$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow*) $\longrightarrow$
p-src p $\in$ (*ipv4cidr-union-set* (*set* (*the* (*ipassmt iface*))))
unfolding *no-spoofing-def*
proof(*intro ballI allI impI*)
fix *iface p*
assume *i*: *iface* $\in$ *dom ipassmt*
and *a*: Γ,γ,*p*(|*p-iiface* := *iface-sel iface*|)$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow*

from *no-spoofing*[*unfolded no-spoofing-def*] *i* **have** *no-spoofing'*:
(*common-matcher*, *in-doubt-allow*),*p*(|*p-iiface* := *iface-sel iface*|)$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalAllow* $\longrightarrow$
p-src p $\in$ *ipv4cidr-union-set* (*set* (*the* (*ipassmt iface*))) **by** *blast*

from *assms simple-imp-good-ruleset FinalAllows-subseteq-in-doubt-allow*[**where** *rs*=*rs*] **have**
{*p*. Γ,γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Decision FinalAllow*} $\subseteq$ {*p*. (*common-matcher*, *in-doubt-allow*),*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalAllow*}
by *blast*
with *a* **have** (*common-matcher*, *in-doubt-allow*),*p*(|*p-iiface* := *iface-sel iface*|)$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalAllow* **by** *blast*
with *no-spoofing'* **show** *p-src p* $\in$ *ipv4cidr-union-set* (*set* (*the* (*ipassmt iface*)))**by** *blast*
qed

context
begin

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-matachAny* :: *'a match-expr* $\Rightarrow$ *bool* **where**
matcheq-matachAny MatchAny $\longleftrightarrow$ *True* |

matcheq-matachAny (*MatchNot m*) $\longleftrightarrow$ ¬ (*matcheq-matachAny m*) |
matcheq-matachAny (*MatchAnd m1 m2*) $\longleftrightarrow$ *matcheq-matachAny m1* ∧ *matcheq-matachAny m2* |
matcheq-matachAny (*Match* -) = *undefined*

private lemma *no-primitives-no-unknown*: ¬ *has-primitive m* $\Longrightarrow$ (*ternary-ternary-eval* (*map-match-tac* β *p m*)) ≠ *TernaryUnknown*
proof(*induction m*)
case *Match* **thus** *?case* **by** *auto*
next
case *MatchAny* **thus** *?case* **by** *simp*
next
case *MatchAnd* **thus** *?case* **by**(*auto elim*: *eval-ternary-And.elims*)
next
case *MatchNot* **thus** *?case* **by**(*auto dest*: *eval-ternary-Not-UnknownD*)
qed

private lemma *no-primitives-matchNot*: **assumes** ¬ *has-primitive m* **shows** *matches* γ (*MatchNot m*) *a p* $\longleftrightarrow$ ¬ *matches* γ *m a p*
proof −
obtain β α **where** (β, α) = γ **by** (*cases* γ, *simp*)
from *assms* **have** *matches* (β, α) (*MatchNot m*) *a p* $\longleftrightarrow$ ¬ *matches* (β, α) *m a p*
apply(*induction m*)
apply(*simp-all add*: *matches-case-ternaryvalue-tuple split*: *ternaryvalue.split*)
apply(*rename-tac m1 m2*)
using *no-primitives-no-unknown* **by** (*metis* (*no-types*, *hide-lams*) *eval-ternary-simps-simple*(*1*) *eval-ternary-simps-simple*(*3*) *ternaryvalue.exhaust*)
with ‹(β, α) = γ› *assms* **show** *?thesis* **by** *simp*
qed

lemma *matcheq-matachAny*: ¬ *has-primitive m* $\Longrightarrow$ *matcheq-matachAny m* $\longleftrightarrow$ *matches* γ *m a p*
proof(*induction m*)
case *Match* **hence** *False* **by** *auto*
thus *?case* **..**
next
case (*MatchNot m*)
from *MatchNot.prems* **have** ¬ *has-primitive m* **by** *simp*
with *no-primitives-matchNot* **have** *matches* γ (*MatchNot m*) *a p* = (¬ *matches* γ *m a p*) **by** *metis*
with *MatchNot* **show** *?case* **by**(*simp*)
next
case (*MatchAnd m1 m2*)
thus *?case* **by**(*simp add*: *Matching-Ternary.bunch-of-lemmata-about-matches*)
next
case *MatchAny* **show** *?case* **by**(*simp add*: *Matching-Ternary.bunch-of-lemmata-about-matches*)

qed
end

context
begin

The set of any ip addresses which may match for a fixed *iface* (overapproximation)

private definition *get-exists-matching-src-ips* :: *iface* ⇒ *common-primitive match-expr* ⇒ *ipv4addr set* **where**
get-exists-matching-src-ips 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
UNIV
else
⋂ *ips* ∈ *set* (*ip-matches*). (*case ips of Pos ip* ⇒ *ipv4s-to-set ip* | *Neg ip* ⇒ − *ipv4s-to-set ip*))
else
{}

value(*code*) *primitive-extractor* (*is-Src*, *src-sel*) (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *30*))) (*Match* (*IIface* (*Iface* ''*eth0*''))))

private lemma *match-simplematcher-Src-getPos*: (∀ *m*∈*set* (*map Src* (*getPos ip-matches*)). *matches* (*common-matcher*, α) (*Match m*) *a p*)
⟷ (∀ *ip*∈*set* (*getPos ip-matches*). *p-src p* ∈ *ipv4s-to-set ip*)
by(*simp add*: *Common-Primitive-Matcher*.*match-simplematcher-SrcDst*)
private lemma *match-simplematcher-Src-getNeg*: (∀ *m*∈*set* (*map Src* (*getNeg ip-matches*)). *matches* (*common-matcher*, α) (*MatchNot* (*Match m*)) *a p*)
⟷ (∀ *ip*∈*set* (*getNeg ip-matches*). *p-src p* ∈ − *ipv4s-to-set ip*)
by(*simp add*: *match-simplematcher-SrcDst-not*)
private lemma *match-simplematcher-Iface-getPos*: (∀ *m*∈*set* (*map IIface* (*getPos i-matches*)). *matches* (*common-matcher*, α) (*Match m*) *a p*)
⟷ (∀ *i*∈*set* (*getPos i-matches*). *match-iface i* (*p-iiface p*))
by(*simp add*: *match-simplematcher-Iface*)
private lemma *match-simplematcher-Iface-getNeg*: (∀ *m*∈*set* (*map IIface* (*getNeg i-matches*)). *matches* (*common-matcher*, α) (*MatchNot* (*Match m*)) *a p*)
⟷ (∀ *i*∈*set* (*getNeg i-matches*). ¬ *match-iface i* (*p-iiface p*))
by(*simp add*: *match-simplematcher-Iface-not*)

private lemma *get-exists-matching-src-ips-subset*:
assumes *normalized-nnf-match m*
shows $\{ip.\ (\exists\, p.\ matches\ (common\text{-}matcher,\ in\text{-}doubt\text{-}allow)\ m\ a\ (p(\!|p\text{-}iiface{:=}\ iface\text{-}sel\ iface,\ p\text{-}src{:=}\ ip|\!)))\} \subseteq$
get-exists-matching-src-ips iface m
proof −

let $?\gamma=(common\text{-}matcher,\ in\text{-}doubt\text{-}allow)$

{ fix *ip-matches p rest src-ip i-matches rest2*
assume *a1*: *primitive-extractor* (*is-Src*, *src-sel*) *m* = (*ip-matches*, *rest*)
and *a2*: *matches* $?\gamma$ *m a* $(p(\!|p\text{-}iiface := iface\text{-}sel\ iface,\ p\text{-}src := src\text{-}ip|\!))$
let $?p=(p(\!|p\text{-}iiface := iface\text{-}sel\ iface,\ p\text{-}src := src\text{-}ip|\!))$

from *primitive-extractor-correct*(*1*)[*OF assms wf-disc-sel-common-primitive*(*3*) *a1*] **have**
$\bigwedge p.\ matches\ ?\gamma\ (alist\text{-}and\ (NegPos\text{-}map\ Src\ ip\text{-}matches))\ a\ p \wedge$
$matches\ ?\gamma\ rest\ a\ p \longleftrightarrow$
$matches\ ?\gamma\ m\ a\ p$ **by** *fast*
with *a2* **have** $matches\ ?\gamma\ (alist\text{-}and\ (NegPos\text{-}map\ Src\ ip\text{-}matches))\ a\ ?p \wedge$
$matches\ ?\gamma\ rest\ a\ ?p$ **by** *simp*
hence $matches\ ?\gamma\ (alist\text{-}and\ (NegPos\text{-}map\ Src\ ip\text{-}matches))\ a\ ?p$ **by** *blast*
with *Negation-Type-Matching.matches-alist-and* **have**
$(\forall\, m{\in}set\ (getPos\ (NegPos\text{-}map\ Src\ ip\text{-}matches)).\ matches\ ?\gamma\ (Match\ m)\ a\ ?p) \wedge$
$(\forall\, m{\in}set\ (getNeg\ (NegPos\text{-}map\ Src\ ip\text{-}matches)).\ matches\ ?\gamma\ (MatchNot\ (Match\ m))\ a\ ?p)$ **by** *metis*
with *getPos-NegPos-map-simp2 getNeg-NegPos-map-simp2* **have**
$(\forall\, m{\in}set\ (map\ Src\ (getPos\ ip\text{-}matches)).\ matches\ ?\gamma\ (Match\ m)\ a\ ?p) \wedge$
$(\forall\, m{\in}set\ (map\ Src\ (getNeg\ ip\text{-}matches)).\ matches\ ?\gamma\ (MatchNot\ (Match\ m))\ a\ ?p)$ **by** *metis*
with *match-simplematcher-Src-getPos match-simplematcher-Src-getNeg* **have** *inset*:
$(\forall\, ip{\in}set\ (getPos\ ip\text{-}matches).\ p\text{-}src\ ?p \in ipv4s\text{-}to\text{-}set\ ip) \wedge (\forall\, ip{\in}set\ (getNeg\ ip\text{-}matches).\ p\text{-}src\ ?p \in -\ ipv4s\text{-}to\text{-}set\ ip)$ **by** *presburger*

with *inset* **have** $\forall\, x \in set\ ip\text{-}matches.\ src\text{-}ip \in (case\ x\ of\ Pos\ x \Rightarrow ipv4s\text{-}to\text{-}set\ x \mid Neg\ ip \Rightarrow -\ ipv4s\text{-}to\text{-}set\ ip)$
apply(*simp add*: *split*: *negation-type.split*)
apply(*safe*)
using *NegPos-set* **apply** *fast*+
done
} note *1=this*

{ fix *ip-matches p rest src-ip i-matches rest2*
assume *a2*: *matches* $?\gamma$ *m a* $(p(\!|p\text{-}iiface := iface\text{-}sel\ iface,\ p\text{-}src := src\text{-}ip|\!))$
and *a4*: *primitive-extractor* (*is-Iiface*, *iiface-sel*) *m* = (*i-matches*, *rest2*)

let *?p*=(*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *src-ip*⦈)

from *primitive-extractor-correct*(*1*)[*OF assms wf-disc-sel-common-primitive*(*5*) *a4*] **have**
⋀*p*. *matches ?γ* (*alist-and* (*NegPos-map IIface i-matches*)) *a p* ∧
matches ?γ rest2 a p ⟷
matches ?γ m a p **by** *fast*
with *a2* **have** *matches ?γ* (*alist-and* (*NegPos-map IIface i-matches*)) *a ?p* ∧
matches ?γ rest2 a ?p **by** *simp*
hence *matches ?γ* (*alist-and* (*NegPos-map IIface i-matches*)) *a ?p* **by** *blast*
with *matches-alist-and* **have**
(∀ *m*∈*set* (*getPos* (*NegPos-map IIface i-matches*)). *matches ?γ* (*Match m*) *a ?p*) ∧
(∀ *m*∈*set* (*getNeg* (*NegPos-map IIface i-matches*)). *matches ?γ* (*MatchNot* (*Match m*)) *a ?p*) **by** *metis*
with *getPos-NegPos-map-simp2 getNeg-NegPos-map-simp2* **have**
(∀ *m*∈*set* (*map IIface* (*getPos i-matches*)). *matches ?γ* (*Match m*) *a ?p*) ∧
(∀ *m*∈*set* (*map IIface* (*getNeg i-matches*)). *matches ?γ* (*MatchNot* (*Match m*)) *a ?p*) **by** *metis*
with *match-simplematcher-Iface-getPos match-simplematcher-Iface-getNeg* **have** *inset-iface*:
(∀ *i*∈*set* (*getPos i-matches*). *match-iface i* (*p-iiface ?p*)) ∧ (∀ *i*∈*set* (*getNeg i-matches*). ¬ *match-iface i* (*p-iiface ?p*)) **by** *presburger*
hence *2*: (∀ *x*∈*set i-matches*. *case x of Pos i* ⇒ *match-iface i* (*iface-sel iface*) | *Neg i* ⇒ ¬ *match-iface i* (*iface-sel iface*))
apply(*simp add*: *split*: *negation-type.split*)
apply(*safe*)
using *NegPos-set* **apply** *fast*+
done
} **note** *2*=*this*

from *1 2* **show** *?thesis*
unfolding *get-exists-matching-src-ips-def*
by(*clarsimp*)
qed

The set of ip addresses which definitely match for a fixed *iface* (underapproximation)

private definition *get-all-matching-src-ips* :: *iface* ⇒ *common-primitive match-expr* ⇒ *ipv4addr set* **where**
get-all-matching-src-ips 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-disc is-Dst rest2* ∧
¬ *has-disc is-Oiface rest2* ∧

¬ *has-disc is-Prot rest2* ∧
¬ *has-disc is-Src-Ports rest2* ∧
¬ *has-disc is-Dst-Ports rest2* ∧
¬ *has-disc is-Extra rest2* ∧
matcheq-matachAny rest2
then
if ip-matches = []
then
UNIV
else
$\bigcap$ *ips* ∈ *set* (*ip-matches*). (*case ips of Pos ip* ⇒ *ipv4s-to-set ip* | *Neg ip* ⇒ − *ipv4s-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* ⊆ {*ip*. (∀ *p*. *matches* (*common-matcher*, *in-doubt-allow*) *m a* (*p*(|*p-iiface*:= *iface-sel iface*, *p-src*:= *ip*|)))}
proof
fix *ip*
assume *a*: *ip* ∈ *get-all-matching-src-ips iface m*
obtain *i-matches rest1* **where** *select1*: *primitive-extractor* (*is-Iiface*, *iiface-sel*) *m* = (*i-matches*, *rest1*) **by** *fastforce*
show *ip* ∈ {*ip*. ∀ *p*. *matches* (*common-matcher*, *in-doubt-allow*) *m a* (*p*(|*p-iiface* := *iface-sel iface*, *p-src* := *ip*|))}
proof(*cases* ∀ *is* ∈ *set i-matches*. (*case is of Pos i* ⇒ *match-iface i* (*iface-sel iface*) | *Neg i* ⇒ ¬*match-iface i* (*iface-sel iface*)))
case *False*
have *get-all-matching-src-ips iface m* = {}
unfolding *get-all-matching-src-ips-def*
using *select1 False* **by** *auto*
with *a* **show** *?thesis* **by** *simp*
next
case *True*
let *?γ*=(*common-matcher*, *in-doubt-allow*)
let *?p*=λ*p*. *p*(|*p-iiface* := *iface-sel iface*, *p-src* := *ip*|)
obtain *ip-matches rest2* **where** *select2*: *primitive-extractor* (*is-Src*, *src-sel*) *rest1* = (*ip-matches*, *rest2*) **by** *fastforce*

let *?noDisc*=¬ *has-disc is-Dst rest2* ∧
¬ *has-disc is-Oiface rest2* ∧
¬ *has-disc is-Prot rest2* ∧
¬ *has-disc is-Src-Ports rest2* ∧ ¬ *has-disc is-Dst-Ports rest2* ∧ ¬ *has-disc is-Extra rest2*

have *get-all-matching-src-ips-caseTrue*: *get-all-matching-src-ips iface m* = (*if ?noDisc* ∧ *matcheq-matachAny rest2*
then if ip-matches = [] *then UNIV else INTER* (*set ip-matches*) (*case-negation-type ipv4s-to-set* (*λip.* − *ipv4s-to-set ip*)) *else* {})
unfolding *get-all-matching-src-ips-def*
by(*simp add*: *True select1 select2*)

from *True* **have** ⋀*p*. (∀ *m*∈*set* (*getPos i-matches*). *matches* (*common-matcher*, *in-doubt-allow*) (*Match* (*IIface m*)) *a* (*?p p*)) ∧
(∀ *m*∈*set* (*getNeg i-matches*). *matches* (*common-matcher*, *in-doubt-allow*) (*MatchNot* (*Match* (*IIface m*))) *a* (*?p p*))
by(*simp add*: *negation-type-forall-split match-simplematcher-Iface match-simplematcher-Iface-not*)
hence *matches-iface*: ⋀*p*. *matches ?γ* (*alist-and* (*NegPos-map IIface i-matches*)) *a* (*?p p*)
by(*simp add*: *matches-alist-and NegPos-map-simps*)

show *?thesis*
proof(*cases ?noDisc* ∧ *matcheq-matachAny rest2*)
case *False*
assume *F*: ¬ (*?noDisc* ∧ *matcheq-matachAny rest2*)
with *get-all-matching-src-ips-caseTrue* **have** *get-all-matching-src-ips iface m* = {} **by** *presburger*
with *a* **have** *False* **by** *simp*
thus *ip* ∈ {*ip*. ∀ *p*. *matches* (*common-matcher*, *in-doubt-allow*) *m a* (*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *ip*⦈)} **..**
next
case *True*
assume *F*: *?noDisc* ∧ *matcheq-matachAny rest2*
with *get-all-matching-src-ips-caseTrue* **have** *get-all-matching-src-ips iface m* =
(*if ip-matches* = [] *then UNIV else INTER* (*set ip-matches*) (*case-negation-type ipv4s-to-set* (*λip.* − *ipv4s-to-set ip*))) **by** *presburger*

from *primitive-extractor-correct*[*OF assms wf-disc-sel-common-primitive*(*5*) *select1*] **have**
select1-matches: ⋀*p*. *matches ?γ* (*alist-and* (*NegPos-map IIface i-matches*)) *a p* ∧ *matches ?γ rest1 a p* ⟷ *matches ?γ m a p*
and *normalized1*: *normalized-nnf-match rest1*
and *no-iiface-rest1*: ¬ *has-disc is-Iiface rest1*
apply −
apply *fast*+
done
from *select1-matches matches-iface* **have** *rest1-matches*: ⋀*p*. *matches ?γ rest1 a* (*?p p*) ⟷ *matches ?γ m a* (*?p p*) **by** *blast*

from *primitive-extractor-correct*[*OF normalized1 wf-disc-sel-common-primitive*(*3*) *select2*] **have**

select2-matches: $\bigwedge p$. (*matches ?γ* (*alist-and* (*NegPos-map Src ip-matches*)) *a p* ∧ *matches ?γ rest2 a p*) = *matches ?γ rest1 a p*
and *no-Src-rest2*: ¬ *has-disc is-Src rest2*
and *no-IIface-rest2*: ¬ *has-disc is-Iiface rest2*
apply −
apply *fast*+
using *no-iiface-rest1* **apply** *fast*
done

from *F* **have** *?noDisc* **by** *simp*
with *no-Src-rest2 no-IIface-rest2* **have** ¬ *has-primitive rest2*
apply(*induction rest2*)
apply(*simp-all*)
apply(*rename-tac x*)
apply(*case-tac x*, *auto*)
done
with *F matcheq-matachAny* **have** $\bigwedge p$. *matches ?γ rest2 a p* **by** *metis*
with *select2-matches rest1-matches* **have** *ip-src-matches*:
$\bigwedge p$. *matches ?γ* (*alist-and* (*NegPos-map Src ip-matches*)) *a* (*?p p*) ⟷ *matches ?γ m a* (*?p p*) **by** *simp*

have *case-nil*: $\bigwedge p$. *ip-matches* = [] ⟹ *matches ?γ* (*alist-and* (*NegPos-map Src ip-matches*)) *a p* **by**(*simp add*: *bunch-of-lemmata-about-matches*)

have *case-list*: $\bigwedge p$. ∀ *x*∈*set ip-matches*. (*case x of Pos i* ⇒ *ip* ∈ *ipv4s-to-set i* | *Neg i* ⇒ *ip* ∈ − *ipv4s-to-set i*) ⟹
matches ?γ (*alist-and* (*NegPos-map Src ip-matches*)) *a* (*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *ip*⦈)
apply(*simp add*: *matches-alist-and NegPos-map-simps*)
apply(*simp add*: *negation-type-forall-split match-simplematcher-SrcDst-not match-simplematcher-SrcDst*)
done

from *a* **show** *ip* ∈ {*ip*. ∀ *p*. *matches* (*common-matcher*, *in-doubt-allow*) *m a* (*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *ip*⦈)}
unfolding *get-all-matching-src-ips-caseTrue*
proof(*clarsimp split*: *split-if-asm*)
fix *p*
assume *ip-matches* = []
with *case-nil* **have** *matches ?γ* (*alist-and* (*NegPos-map Src ip-matches*)) *a* (*?p p*) **by** *simp*
with *ip-src-matches* **show** *matches ?γ m a* (*?p p*) **by** *simp*
next
fix *p*
assume ∀ *x*∈*set ip-matches*. *ip* ∈ (*case x of Pos x* ⇒ *ipv4s-to-set x* | *Neg ip* ⇒ − *ipv4s-to-set ip*)
hence ∀ *x*∈*set ip-matches*. *case x of Pos i* ⇒ *ip* ∈ *ipv4s-to-set i* | *Neg i* ⇒ *ip* ∈ − *ipv4s-to-set i*
by(*simp-all split*: *negation-type.split negation-type.split-asm*)

with *case-list* **have** *matches ?γ* (*alist-and* (*NegPos-map Src ip-matches*)) *a* (*?p p*) **.**
with *ip-src-matches* **show** *matches ?γ m a* (*?p p*) **by** *simp*
qed
qed
qed
qed

private definition *get-exists-matching-src-ips-executable* :: *iface* ⇒ *common-primitive match-expr* ⇒ *32 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
ipv4range-UNIV
else
l2br-negation-type-intersect (*NegPos-map ipt-ipv4range-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*
apply(*simp add*: *get-exists-matching-src-ips-executable-def get-exists-matching-src-ips-def*)
apply(*case-tac primitive-extractor* (*is-Iiface*, *iiface-sel*) *m*)
apply(*case-tac primitive-extractor* (*is-Src*, *src-sel*) *m*)
apply(*simp*)
apply(*simp add*: *l2br-negation-type-intersect*)
apply(*simp add*: *ipv4range-UNIV-def NegPos-map-simps*)
apply(*simp add*: *ipt-ipv4range-to-interval*)
apply(*safe*)
apply(*simp-all add*: *ipt-ipv4range-to-interval*)
apply(*rename-tac i-matches rest1 a b x xa*)
apply(*case-tac xa*)
apply(*simp-all add*: *NegPos-set*)
using *ipt-ipv4range-to-interval* **apply** *fast+*
apply(*rename-tac i-matches rest1 a b x aa ab ba*)
apply(*erule-tac x=Pos aa* **in** *ballE*)
apply(*simp-all add*: *NegPos-set*)
using *NegPos-set*(*2*) **by** *fastforce*

value(*code*) (*get-exists-matching-src-ips-executable* (*Iface ''eth0''*)

(*MatchAnd* (*MatchNot* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *24*))))
(*Match* (*IIface* (*Iface* ''*eth0*'')))))

private definition *get-all-matching-src-ips-executable* :: *iface* ⇒ *common-primitive match-expr* ⇒ *32 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-disc is-Dst rest2* ∧
¬ *has-disc is-Oiface rest2* ∧
¬ *has-disc is-Prot rest2* ∧
¬ *has-disc is-Src-Ports rest2* ∧
¬ *has-disc is-Dst-Ports rest2* ∧
¬ *has-disc is-Extra rest2* ∧
matcheq-matachAny rest2
then
if ip-matches = []
then
ipv4range-UNIV
else
l2br-negation-type-intersect (*NegPos-map ipt-ipv4range-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*
apply(*simp add*: *get-all-matching-src-ips-executable-def get-all-matching-src-ips-def*)
apply(*case-tac primitive-extractor* (*is-Iiface*, *iiface-sel*) *m*)
apply(*simp*, *rename-tac i-matches rest1*)
apply(*case-tac primitive-extractor* (*is-Src*, *src-sel*) *rest1*)
apply(*simp*)
apply(*simp add*: *l2br-negation-type-intersect*)
apply(*simp add*: *ipv4range-UNIV-def NegPos-map-simps*)
apply(*simp add*: *ipt-ipv4range-to-interval*)
apply(*safe*)
apply(*simp-all add*: *ipt-ipv4range-to-interval*)
apply(*rename-tac i-matches rest1 a b x xa*)
apply(*case-tac xa*)
apply(*simp-all add*: *NegPos-set*)
using *ipt-ipv4range-to-interval* **apply** *fast+*
apply(*rename-tac i-matches rest1 a b x aa ab ba*)

```
   apply(erule-tac x=Pos aa in ballE)
    apply(simp-all add: NegPos-set)
  apply(erule-tac x=Neg aa in ballE)
   apply(simp-all add: NegPos-set)
  done
 value(code) (get-all-matching-src-ips-executable (Iface ''eth0'')
     (MatchAnd (MatchNot (Match (Src (Ip4AddrNetmask (192,168,0,0) 24))))
(Match (IIface (Iface ''eth0'')))))
```

```
 private lemma {ip. ∀ p ∈ {p. ¬ match-iface iface (p-iiface p)}. matches (common-matcher,
in-doubt-allow) m Drop (p(| p-src:= ip|))} =
           {ip. ∀ p. ¬ match-iface iface (p-iiface p) ⟶ matches (common-matcher,
in-doubt-allow) m Drop (p(| p-src:= ip|))} by blast
 private lemma p-iiface-update: p(|p-iiface := p-iiface p, p-src := x|) = p(|p-src
:= x|) by(simp)
 private lemma (⋂ if' ∈ {if'. ¬ match-iface iface if'}. {ip. ∀ p. matches (common-matcher,
in-doubt-allow) m Drop (p(| p-iiface := if', p-src:= ip|))}) =
          {ip. ∀ p ∈ {p. ¬ match-iface iface (p-iiface p)}. matches (common-matcher,
in-doubt-allow) m Drop (p(| p-src:= ip|))}
  apply(simp)
  apply(safe)
   apply(simp-all)
  apply(erule-tac x=(p-iiface p) in allE)
  apply(simp)
  using p-iiface-update by metis
```

The following algorithm sound but not complete.

```
 private fun no-spoofing-algorithm :: iface ⇒ ipassignment ⇒ common-primitive
rule list ⇒ ipv4addr set ⇒ ipv4addr set ⇒ (*ipv4addr set ⇒*) bool where
  no-spoofing-algorithm iface ipassmt [] allowed denied1 ⟷
    (allowed − denied1) ⊆ ipv4cidr-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 ⇀ (ipv4addr ×
nat) list) ⇒ common-primitive rule list ⇒ 32 wordinterval ⇒ 32 wordinterval ⇒
bool where
  no-spoofing-algorithm-executable iface ipassmt [] allowed denied1 ⟷
  wordinterval-subset (wordinterval-setminus allowed denied1) (l2br (map ipv4cidr-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 denied1* = *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*)
apply(*induction iface ipassmt rs allowed denied rule*: *no-spoofing-algorithm-executable.induct*)
apply(*simp-all*)
apply(*simp-all add*: *get-exists-matching-src-ips-executable get-all-matching-src-ips-executable*)
apply(*simp add*: *ipv4cidr-union-set-def l2br*)
apply(*subgoal-tac* (⋃ *a*∈*set* (*the* (*ipassmt iface*)). *case ipv4cidr-to-interval a of* (*x*, *xa*) ⇒ {*x*..*xa*}) =
(⋃ *x*∈*set* (*the* (*ipassmt iface*)). *case x of* (*base*, *len*) ⇒ *ipv4range-set-from-bitmask base len*))
apply(*simp-all*)
apply(*safe*)
apply(*simp-all*)
apply(*rule-tac x*=(*a*, *b*) **in** *bexI*)
apply(*simp-all add*: *ipv4cidr-to-interval*)
apply(*rule-tac x*=(*a*, *b*) **in** *bexI*)
apply(*simp-all*)
using *ipv4cidr-to-interval* **by** *blast*

lemma *no-spoofing-algorithm-executable*
(*Iface* ''*eth0*'')
[*Iface* ''*eth0*'' ↦ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *24*))) (*Match* (*IIface* (*Iface* ''*eth0*''))) *action.Accept*,
Rule MatchAny action.Drop]
Empty-WordInterval Empty-WordInterval **by** *eval*

Examples

Example 1

Ruleset: Accept all non-spoofed packets, drop rest.

lemma *no-spoofing-algorithm*
(*Iface* ''*eth0*'')
[*Iface* ''*eth0*'' ↦ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *24*))) (*Match* (*IIface* (*Iface* ''*eth0*''))) *action.Accept*,

Rule MatchAny action.Drop]
{} {}

proof −
have *localrng*: *ipv4range-set-from-bitmask* (*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*)) *24* = {*0xC0A80000*..*0xC0A800FF*}
by(*simp add*: *ipv4range-set-from-bitmask-def ipv4range-set-from-netmask-def ipv4addr-of-dotdecimal.simps ipv4addr-of-nat-def*)

have *ipset*: {*ip*. ∃ *p*. *matches* (*common-matcher*, *in-doubt-allow*) (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*192*, *168*, *0*, *0*) *24*))) (*Match* (*IIface* (*Iface* ''*eth0*'')))) *Accept*
(*p*(|*p-iiface* := ''*eth0*'', *p-src* := *ip*|))} = *ipv4range-set-from-bitmask* (*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*)) *24*
by(*auto simp add*: *localrng eval-ternary-simps bool-to-ternary-simps matches-case-ternaryvalue-tuple match-iface.simps*
split: *ternaryvalue.split ternaryvalue.split-asm*)
show *?thesis*
apply(*simp add*: *ipset ipv4cidr-union-set-def get-exists-matching-src-ips-def*)
by *blast*
qed

Example 2

Ruleset: Drop packets from a spoofed IP range, allow rest.

lemma *no-spoofing-algorithm*
(*Iface* ''*eth0*'')
[*Iface* ''*eth0*'' ↦ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*Match* (*IIface* (*Iface* ''*wlan+*''))) (*Match* (*Extra* ''*no idea what this is*''))) *action.Accept*, (∗*not interesting for spoofing*∗)
Rule (*MatchNot* (*Match* (*IIface* (*Iface* ''*eth0+*'')))) *action.Accept*, (∗*not interesting for spoofing*∗)
Rule (*MatchAnd* (*MatchNot* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *24*)))) (*Match* (*IIface* (*Iface* ''*eth0*'')))) *action.Drop*, (∗*spoof*−*protect here*∗)
Rule MatchAny action.Accept]
{} {}

apply(*simp add*: *get-all-matching-src-ips-def get-exists-matching-src-ips-def match-iface.simps*)
apply(*simp add*: *ipv4cidr-union-set-def*)
done

private lemma *iprange-example*: *ipv4range-set-from-bitmask* (*ipv4addr-of-dotdecimal* (*192*, *168*, *0*, *0*)) *24* = {*0xC0A80000*..*0xC0A800FF*}
by(*simp add*: *ipv4range-set-from-bitmask-def ipv4range-set-from-netmask-def ipv4addr-of-dotdecimal.simps ipv4addr-of-nat-def*)
lemma *no-spoofing-algorithm*
(*Iface* ''*eth0*'')
[*Iface* ''*eth0*'' ↦ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]

[*Rule* (*MatchNot* (*Match* (*IIface* (*Iface* ''*wlan+*'')))) *action.Accept*,
(∗*accidently allow everything for eth0*∗)
Rule (*MatchAnd* (*MatchNot* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *24*)))) (*Match* (*IIface* (*Iface* ''*eth0*'')))) *action.Drop*,
Rule MatchAny action.Accept]
{} {} ⟷ *False*

apply(*simp add*: *get-exists-matching-src-ips-def match-iface.simps*)
apply(*simp add*: *range-0-max-UNIV ipv4cidr-union-set-def iprange-example*)
done

Example 3

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 *no-spoofing* [*Iface* ''*eth0*'' ↦ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *24*))) (*MatchNot* (*Match* (*IIface* (*Iface* ''*eth0*''))))) *action.Drop*,
Rule MatchAny action.Accept] ⟷ *False* (**is** *no-spoofing ?ipassmt ?rs* ⟷ *False*)
proof −
have *simple-ruleset ?rs* **by**(*simp add*: *simple-ruleset-def*)
hence *1*: $\forall$ *p*. (*common-matcher*, *in-doubt-allow*),*p*⊢ ⟨*?rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalAllow* ⟷
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) *p ?rs Undecided* = *Decision FinalAllow*
apply −
apply(*drule simple-imp-good-ruleset*)
apply(*simp add*: *approximating-semantics-iff-fun-good-ruleset*)
done
have *2*: $\forall$ *p*. ¬ *matches* (*common-matcher*, *in-doubt-allow*) (*MatchNot* (*Match* (*IIface* (*Iface* ''*eth0*'')))) *Drop* (*p*⦇*p-iiface* := ''*eth0*''⦈)
by(*simp add*: *match-simplematcher-Iface-not match-iface.simps*)

The *no-spoofing* definition requires that all packets from ''*eth0*'' are from 192.168.0.0/24

have *no-spoofing ?ipassmt ?rs* ⟷
($\forall$ *p*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) (*p*⦇*p-iiface* := ''*eth0*''⦈) *?rs Undecided* = *Decision FinalAllow* ⟶
p-src p ∈ *ipv4cidr-union-set* {(*ipv4addr-of-dotdecimal* (*192*, *168*, *0*, *0*), *24*)})
unfolding *no-spoofing-def*
apply(*subst 1*)
by(*simp*)

In this example however, all packets for all IPs from ''*eth0*'' are allowed.

have ∀ *p. approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) (*p*⦇*p-iiface* := ''*eth0*''⦈) *?rs Undecided* = *Decision FinalAllow*
by(*simp add*: *bunch-of-lemmata-about-matches ternary-to-bool-bool-to-ternary* *2*)

have *3*: *no-spoofing ?ipassmt ?rs* ⟷ (∀ *p*::*simple-packet*. *p-src p* ∈ *ipv4cidr-union-set* {(*ipv4addr-of-dotdecimal* (*192*, *168*, *0*, *0*), *24*)})
unfolding *no-spoofing-def*
apply(*subst 1*)
apply(*simp*)
apply(*simp add*: *bunch-of-lemmata-about-matches ternary-to-bool-bool-to-ternary*)
apply(*simp add*: *2*)
done

show *?thesis*
unfolding *3*
apply(*simp add*: *ipv4cidr-union-set-def*)
apply(*simp add*: *iprange-example*)
apply(*rule-tac x*=*p*⦇*p-src* := *0*⦈ **in** *exI*)
apply(*simp*)
done
qed
lemma *no-spoofing-algorithm*
(*Iface* ''*eth0*'')
[*Iface* ''*eth0*'' ↦ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *24*))) (*MatchNot* (*Match* (*IIface* (*Iface* ''*eth0*''))))) *action.Drop*,
Rule MatchAny action.Accept]
{} {} ⟷ *False*
apply(*subst no-spoofing-algorithm.simps*)
apply(*simp add*: *get-exists-matching-src-ips-def get-all-matching-src-ips-def match-iface.simps del*: *no-spoofing-algorithm.simps*)
apply(*subst no-spoofing-algorithm.simps*)
apply(*simp add*: *get-exists-matching-src-ips-def get-all-matching-src-ips-def match-iface.simps del*: *no-spoofing-algorithm.simps*)
apply(*subst no-spoofing-algorithm.simps*)
apply(*simp add*: *get-exists-matching-src-ips-def get-all-matching-src-ips-def match-iface.simps del*: *no-spoofing-algorithm.simps*)
apply(*simp add*: *ipv4cidr-union-set-def*)
apply(*simp add*: *iprange-example*)
apply(*simp add*: *range-0-max-UNIV*)
done

Example 4: spoofing protection but the algorithm fails (it is only sound, not complete).

lemma *no-spoofing* [*Iface* ''*eth0*'' ↦ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*MatchNot* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *24*)))) (*MatchAnd* (*Match* (*IIface* (*Iface* ''*eth0*''))) (*Match* (*Prot* (*Proto TCP*)))))

action.Drop,
Rule (*MatchAnd* (*MatchNot* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *24*)))) (*MatchAnd* (*Match* (*IIface* (*Iface* ''*eth0*''))) (*MatchNot* (*Match* (*Prot* (*Proto* *TCP*)))))) *action.Drop*,
Rule MatchAny action.Accept] (**is** *no-spoofing ?ipassmt ?rs*)
proof −
have *simple-ruleset ?rs* **by**(*simp add*: *simple-ruleset-def*)
hence *1*: $\forall$ *p*. (*common-matcher*, *in-doubt-allow*),*p*$\vdash$ $\langle$*?rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalAllow* $\longleftrightarrow$
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) *p ?rs Undecided* = *Decision FinalAllow*
apply −
apply(*drule simple-imp-good-ruleset*)
apply(*simp add*: *approximating-semantics-iff-fun-good-ruleset*)
done
show *?thesis*
unfolding *no-spoofing-def*
apply(*subst 1*)
apply(*simp*)
apply(*simp add*: *bunch-of-lemmata-about-matches ternary-to-bool-bool-to-ternary*)
apply(*simp add*: *match-iface.simps*)
apply(*simp add*: *match-simplematcher-SrcDst-not*)
apply(*auto simp add*: *eval-ternary-simps bool-to-ternary-simps matches-case-ternaryvalue-tuple match-iface.simps*
split: *ternaryvalue.split ternaryvalue.split-asm*)
apply(*simp add*: *ipv4cidr-union-set-def*)
done
qed
lemma ¬ *no-spoofing-algorithm*
(*Iface* ''*eth0*'')
[*Iface* ''*eth0*'' $\mapsto$ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*MatchNot* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *24*)))) (*MatchAnd* (*Match* (*IIface* (*Iface* ''*eth0*''))) (*Match* (*Prot* (*Proto TCP*))))) *action.Drop*,
Rule (*MatchAnd* (*MatchNot* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *24*)))) (*MatchAnd* (*Match* (*IIface* (*Iface* ''*eth0*''))) (*MatchNot* (*Match* (*Prot* (*Proto* *TCP*)))))) *action.Drop*,
Rule MatchAny action.Accept] {} {}
by(*simp add*: *get-exists-matching-src-ips-def get-all-matching-src-ips-def match-iface.simps ipv4cidr-union-set-def iprange-example range-0-max-UNIV*)

private definition *nospoof iface ipassmt rs* = ($\forall$ *p*.
(*approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) (*p*(|*p-iiface*:=*iface-sel iface*|)) *rs Undecided* = *Decision FinalAllow*) $\longrightarrow$
p-src p $\in$ (*ipv4cidr-union-set* (*set* (*the* (*ipassmt iface*)))))
private definition *setbydecision iface rs dec* = {*ip*. $\exists$ *p*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*)
(*p*(|*p-iiface*:=*iface-sel iface*, *p-src* := *ip*|)) *rs Undecided* = *Decision dec*}

private lemma *packet-update-iface-simp*: *p*(|*p-iiface* := *iface-sel iface*, *p-src* := *x*|) = *p*(|*p-src* := *x*, *p-iiface* := *iface-sel iface*|) **by** *simp*

private lemma *nospoof-setbydecision*: *nospoof iface ipassmt rs* ⟷ *setbydecision iface rs FinalAllow* ⊆ (*ipv4cidr-union-set* (*set* (*the* (*ipassmt iface*))))
proof
assume *a*: *nospoof iface ipassmt rs*
from *a* **show** *setbydecision iface rs FinalAllow* ⊆ *ipv4cidr-union-set* (*set* (*the* (*ipassmt iface*)))
apply(*simp add*: *nospoof-def setbydecision-def*)
apply(*safe*)
apply(*rename-tac x p*)
apply(*erule-tac x*=*p*(|*p-iiface* := *iface-sel iface*, *p-src* := *x*|) **in** *allE*)
apply(*simp*)
apply(*simp add*: *packet-update-iface-simp*)
done
next
assume *a1*: *setbydecision iface rs FinalAllow* ⊆ *ipv4cidr-union-set* (*set* (*the* (*ipassmt iface*)))
show *nospoof iface ipassmt rs*
unfolding *nospoof-def*
proof(*safe*)
fix *p*
assume *a2*: *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) (*p*(|*p-iiface* := *iface-sel iface*|)) *rs Undecided* = *Decision FinalAllow*
— In *setbydecision-fix-p*the ∃ quantifier is gone and we consider this set for *p*.
let *?setbydecision-fix-p*={*ip*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) (*p*(|*p-iiface* := *iface-sel iface*, *p-src* := *ip*|)) *rs Undecided* = *Decision FinalAllow*}
from *a1 a2* **have** *1*: *?setbydecision-fix-p* ⊆ *ipv4cidr-union-set* (*set* (*the* (*ipassmt iface*))) **by**(*simp add*: *nospoof-def setbydecision-def*) *blast*
from *a2* **have** *2*: *p-src p* ∈ *?setbydecision-fix-p* **by** *simp*
from *1 2* **show** *p-src p* ∈ *ipv4cidr-union-set* (*set* (*the* (*ipassmt iface*))) **by** *blast*
qed
qed

private definition *setbydecision-all iface rs dec* = {*ip*. ∀ *p*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*)
(*p*(|*p-iiface*:=*iface-sel iface*, *p-src* := *ip*|)) *rs Undecided* = *Decision dec*}

private lemma *setbydecision-setbydecision-all-Allow*: (*setbydecision iface rs FinalAllow* − *setbydecision-all iface rs FinalDeny*) =
setbydecision iface rs FinalAllow
apply(*safe*)

apply(*simp add*: *setbydecision-def setbydecision-all-def*)
done
private lemma *setbydecision-setbydecision-all-Deny*: (*setbydecision iface rs FinalDeny* − *setbydecision-all iface rs FinalAllow*) =
setbydecision iface rs FinalDeny
apply(*safe*)
apply(*simp add*: *setbydecision-def setbydecision-all-def*)
done

private lemma *decision-append*: *simple-ruleset rs1* ⟹ *approximating-bigstep-fun* γ *p rs1 Undecided* = *Decision X* ⟹
approximating-bigstep-fun γ *p* (*rs1* @ *rs2*) *Undecided* = *Decision X*
apply(*drule simple-imp-good-ruleset*)
apply(*drule good-imp-wf-ruleset*[*of* - γ *p*])
apply(*simp add*: *approximating-bigstep-fun-seq-wf Decision-approximating-bigstep-fun*)
done

private lemma *setbydecision-append*: *simple-ruleset* (*rs1* @ *rs2*) ⟹ *setbydecision iface* (*rs1* @ *rs2*) *FinalAllow* =
setbydecision iface rs1 FinalAllow ∪ {*ip*. ∃ *p*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*)
(*p*⦇*p-iiface*:=*iface-sel iface*, *p-src* := *ip*⦈) *rs2 Undecided* = *Decision FinalAllow* ∧
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) (*p*⦇*p-iiface*:=*iface-sel iface*, *p-src* := *ip*⦈) *rs1 Undecided* = *Undecided*}
apply(*simp add*: *setbydecision-def*)
apply(*subst Set.Collect-disj-eq*[*symmetric*])
apply(*rule Set.Collect-cong*)
apply(*subst approximating-bigstep-fun-seq-Undecided-t-wf*)
apply(*simp add*: *simple-imp-good-ruleset good-imp-wf-ruleset*)
by *blast*

private lemma *not-FinalAllow*: *foo* ≠ *Decision FinalAllow* ⟷ *foo* = *Decision FinalDeny* ∨ *foo* = *Undecided*
apply(*cases foo*)
apply *simp-all*
apply(*rename-tac x2*)
apply(*case-tac x2*)
apply(*simp-all*)
done

private lemma *foo-not-FinalDeny*: *foo* ≠ *Decision FinalDeny* ⟷ *foo* = *Undecided* ∨ *foo* = *Decision FinalAllow*
apply(*cases foo*)
apply *simp-all*
apply(*rename-tac x2*)
apply(*case-tac x2*)
apply(*simp-all*)

done

private lemma *setbydecision-all-appendAccept*: *simple-ruleset* (*rs* @ [*Rule r Accept*]) $\Longrightarrow$
setbydecision-all iface rs FinalDeny = *setbydecision-all iface* (*rs* @ [*Rule r Accept*]) *FinalDeny*
apply(*simp add*: *setbydecision-all-def*)
apply(*rule Set.Collect-cong*)
apply(*subst approximating-bigstep-fun-seq-Undecided-t-wf*)
apply(*simp add*: *simple-imp-good-ruleset good-imp-wf-ruleset*)
apply(*simp add*: *not-FinalAllow*)
done

lemma $(\forall x.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall x.\ P\ x) \wedge (\forall x.\ Q\ x)$ **by** *blast*
lemma $(\forall x.\ P\ x) \vee (\forall x.\ Q\ x) \Longrightarrow (\forall x.\ P\ x \vee Q\ x)$ **by** *blast*

private lemma *setbydecision-all-append-subset*: *simple-ruleset* (*rs1* @ *rs2*) $\Longrightarrow$
setbydecision-all iface rs1 FinalDeny ∪ {*ip*. ∀ *p*.
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) (*p*(|*p-iiface*:=*iface-sel iface*, *p-src* := *ip*|)) *rs2 Undecided* = *Decision FinalDeny* ∧
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) (*p*(|*p-iiface*:=*iface-sel iface*, *p-src* := *ip*|)) *rs1 Undecided* = *Undecided*}
⊆
setbydecision-all iface (*rs1* @ *rs2*) *FinalDeny*
unfolding *setbydecision-all-def*
apply(*subst Set.Collect-disj-eq*[*symmetric*])
apply(*rule Set.Collect-mono*)
apply(*subst approximating-bigstep-fun-seq-Undecided-t-wf*)
apply(*simp add*: *simple-imp-good-ruleset good-imp-wf-ruleset*)
apply(*simp add*: *not-FinalAllow*)
done

private lemma *Collect-minus-eq*: $\{x.\ P\ x\} - \{x.\ Q\ x\} = \{x.\ P\ x \wedge \neg\ Q\ x\}$ **by** *blast*

private lemma *setbydecision-all iface rs1 FinalDeny* ∪
{*ip*. ∀ *p*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) (*p*(|*p-iiface* := *iface-sel iface*, *p-src* := *ip*|)) *rs1 Undecided* = *Undecided*}
⊆
− *setbydecision iface rs1 FinalAllow*
unfolding *setbydecision-all-def*
unfolding *setbydecision-def*
apply(*subst Set.Collect-neg-eq*[*symmetric*])
apply(*subst Set.Collect-disj-eq*[*symmetric*])
apply(*rule Set.Collect-mono*)
by(*simp*)

private lemma *setbydecision-all-append-subset2*:

simple-ruleset (*rs1* @ *rs2*) $\Longrightarrow$ *setbydecision-all iface rs1 FinalDeny* $\cup$ (*setbydecision-all iface rs2 FinalDeny* $-$ *setbydecision iface rs1 FinalAllow*)
$\subseteq$ *setbydecision-all iface* (*rs1* @ *rs2*) *FinalDeny*
unfolding *setbydecision-all-def*
unfolding *setbydecision-def*
apply(*subst Collect-minus-eq*)
apply(*subst Set.Collect-disj-eq*[*symmetric*])
apply(*rule Set.Collect-mono*)
apply(*subst approximating-bigstep-fun-seq-Undecided-t-wf*)
apply(*simp add*: *simple-imp-good-ruleset good-imp-wf-ruleset*)
apply(*intro impI allI*)
apply(*simp add*: *not-FinalAllow*)
apply(*case-tac approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) (*p*(|*p-iiface* := *iface-sel iface*, *p-src* := *x*|)) *rs1 Undecided*)
apply(*elim disjE*)
apply(*simp-all*)[*2*]
apply(*rename-tac x2*)
apply(*case-tac x2*)
prefer *2*
apply *simp*
apply(*elim disjE*)
apply(*simp*)
by *blast*

private lemma *notin-setbydecisionD*: *ip* $\notin$ *setbydecision iface rs FinalAllow* $\Longrightarrow$ ($\forall$ *p*.
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) (*p*(|*p-iiface*:=*iface-sel iface*, *p-src* := *ip*|)) *rs Undecided* = *Decision FinalDeny* $\vee$
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) (*p*(|*p-iiface*:=*iface-sel iface*, *p-src* := *ip*|)) *rs Undecided* = *Undecided*)
by(*simp add*: *setbydecision-def not-FinalAllow*)

private lemma *helper1*: $a \wedge (a \longrightarrow b) \longleftrightarrow a \wedge b$ **by** *auto*

private lemma *simple-ruleset* (*rs* @ [*Rule r Accept*]) $\Longrightarrow$
(*setbydecision iface rs FinalAllow* $\cup$ {*ip*. $\exists$ *p*. *matches* (*common-matcher*, *in-doubt-allow*) *r Accept* (*p*(|*p-iiface* := *iface-sel iface*, *p-src* := *ip*|))}) =
setbydecision iface (*rs* @ [*Rule r Accept*]) *FinalAllow*
apply(*simp add*: *setbydecision-append*)
apply(*simp add*: *helper1*)
apply(*rule*)
prefer *2*
apply *blast*
apply(*simp*)
apply(*safe*)
apply(*drule notin-setbydecisionD*)
apply(*rule-tac x*=*p* **in** *exI*)

apply(*simp*)

oops

private lemma $- \{ip.\ \exists\, p.\ \neg$ *match-iface iface* (*p-iiface p*) $\vee \neg$ *matches* (*common-matcher*, *in-doubt-allow*) *m Drop* ($p(\!|p\text{-}src := ip|\!)$)}
$\subseteq$ *setbydecision-all iface* ([*Rule m Drop*]) *FinalDeny*
apply(*simp add*: *setbydecision-all-def*)
apply(*subst Collect-neg-eq*[*symmetric*])
apply(*rule Set.Collect-mono*)
apply(*simp*)
done

private lemma *setbydecision-all-not-iface*: ($\bigcap$ *if'* $\in$ {*if'*. $\neg$ *match-iface iface if'*}. *setbydecision-all* (*Iface if'*) *rs1 FinalDeny*) =
{*ip*. $\forall\, p$. $\neg$ *match-iface iface* (*p-iiface p*) $\longrightarrow$
approximating-bigstep-fun (*common-matcher*, *in-doubt-allow*) ($p(\!|p\text{-}src := ip|\!)$) *rs1 Undecided* = *Decision FinalDeny*}
apply(*simp add*: *setbydecision-all-def*)
apply(*safe*)
apply(*simp-all*)
apply(*erule-tac x*=(*p-iiface p*) **in** *allE*)
apply(*simp*)
using *p-iiface-update* **by** *metis*

private lemma *setbydecision-all2*: *setbydecision-all iface rs dec* =
{*ip*. $\forall\, p$. (*iface-sel iface*) = (*p-iiface p*) $\longrightarrow$ *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) ($p(\!|p\text{-}src := ip|\!)$) *rs Undecided* = *Decision dec*}
apply(*simp add*: *setbydecision-all-def*)
apply(*rule Set.Collect-cong*)
apply(*rule iffI*)
apply(*clarify*)
apply(*erule-tac x*=*p* **in** *allE*)
apply(*simp*)
apply(*clarify*)
apply(*erule-tac x*=$p(\!|p\text{-}iiface := iface\text{-}sel\ iface|\!)$ **in** *allE*)
apply(*simp*)
done

private lemma {*ip*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) ($p(\!|p\text{-}src := ip|\!)$) *rs Undecided* = *Decision dec*} =
{*ip* | *ip*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) ($p(\!|p\text{-}src := ip|\!)$) *rs Undecided* = *Decision dec*} **by** *simp*

private lemma *setbydecision-all2'*: *setbydecision-all iface rs dec* =
{*ip*. $\forall\, p$. (*iface-sel iface*) = (*p-iiface p*) $\longrightarrow$ *p-src p* = *ip* $\longrightarrow$ *approximating-bigstep-fun*

(*common-matcher*, *in-doubt-allow*) *p rs Undecided* = *Decision dec*}
  apply(*simp add*: *setbydecision-all-def*)
  apply(*rule Set.Collect-cong*)
  apply(*rule iffI*)
   apply(*clarify*)
   apply(*erule-tac x*=*p* **in** *allE*)
   apply(*simp*)
  apply(*clarify*)
  apply(*erule-tac x*=*p*(|*p-iiface* := *iface-sel iface*, *p-src* := *ip*|) **in** *allE*)
  apply(*simp*)
  done

private lemma *setbydecision-all3*: *setbydecision-all iface rs dec* = ($\bigcap$ *p* ∈ {*p*. (*iface-sel iface*) = (*p-iiface p*)}.
    {*ip*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) (*p*(|*p-src* := *ip*|)) *rs Undecided* = *Decision dec*})
  apply(*simp add*: *setbydecision-all2*)
  by *blast*

private lemma *setbydecision-all3'*: *setbydecision-all iface rs dec* = ($\bigcap$ *p* ∈ {*p*. (*iface-sel iface*) = (*p-iiface p*)}.
    {*ip* | *ip*. *p-src p* = *ip* ⟶ *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) *p rs Undecided* = *Decision dec*})
  apply(*simp add*: *setbydecision-all3*)
  apply(*safe*)
  apply(*simp-all*)
  by *fastforce*

private lemma *setbydecision-all4*: *setbydecision-all iface rs dec* =
  ($\bigcap$ *p* ∈ {*p*. ¬ *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) *p rs Undecided* = *Decision dec*}.
    {*ip*. *p-src p* = *ip* ⟶ (*iface-sel iface*) ≠ (*p-iiface p*)})
  apply(*simp add*: *setbydecision-all2'*)
  apply(*safe*)
  apply(*simp-all*)
  by *blast*

private lemma *setbydecision2*: *setbydecision iface rs dec* =
  {*ip*. ∃ *p*. (*iface-sel iface*) = (*p-iiface p*) ∧ *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) (*p*(|*p-src* := *ip*|)) *rs Undecided* = *Decision dec*}
  apply(*simp add*: *setbydecision-def*)
  apply(*rule Set.Collect-cong*)
  apply(*rule iffI*)
   apply(*clarify*)
   apply(*rule-tac x*=*p*(|*p-iiface* := *iface-sel iface*|) **in** *exI*)
   apply(*simp*)

apply(*clarify*)
apply(*rule-tac x=p* **in** *exI*)
apply(*simp*)
done

private lemma *setbydecision3*: *setbydecision iface rs dec* = ($\bigcup$ *p* ∈ {*p*. (*iface-sel iface*) = (*p-iiface p*)}.
{*ip*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) (*p*⦇*p-src* := *ip*⦈) *rs Undecided* = *Decision dec*})
apply(*simp add*: *setbydecision2*)
by *blast*

private lemma {*ip*. (*iface-sel iface*) = (*p-iiface p*) ∧ *p-src p* = *ip*} = {*ip* | *ip*. (*iface-sel iface*) = (*p-iiface p*) ∧ *p-src p* = *ip*} **by** *simp*

private lemma *setbydecision4*: *setbydecision iface rs dec* =
($\bigcup$ *p* ∈ {*p*. *approximating-bigstep-fun* (*common-matcher*, *in-doubt-allow*) *p rs Undecided* = *Decision dec*}.
{*ip*. (*iface-sel iface*) = (*p-iiface p*) ∧ *p-src p* = *ip*})
apply(*simp add*: *setbydecision2*)
by *fastforce*

private lemma *setbydecision-all iface rs FinalDeny* ⊆ − *setbydecision iface rs FinalAllow*
apply(*simp add*: *setbydecision-def setbydecision-all-def*)
apply(*subst Set.Collect-neg-eq*[*symmetric*])
apply(*rule Set.Collect-mono*)
apply(*simp*)
done

lemma *a* − (*d1* ∪ (*d2* − *a*)) = *a* − *d1* **by** *auto*

private lemma *xxhlpsubset1*: *Y* ⊆ *X* ⟹ ∀ *x*. *S x* ⊆ *S′ x* ⟹ ($\bigcap$*x*∈*X*. *S x*) ⊆ ($\bigcap$*x*∈*Y*. *S′ x*) **by** *auto*
private lemma *xxhlpsubset2*: *X* ⊆ *Y* ⟹ ∀ *x*. *S x* ⊆ *S′ x* ⟹ ($\bigcup$*x*∈*X*. *S x*) ⊆ ($\bigcup$*x*∈*Y*. *S′ x*) **by** *auto*

private lemma *no-spoofing-algorithm-sound-generalized*:
shows *simple-ruleset rs1* ⟹ *simple-ruleset rs2* ⟹
(∀ *r* ∈ *set rs2*. *normalized-nnf-match* (*get-match r*)) ⟹
setbydecision iface rs1 FinalAllow ⊆ *allowed* ⟹
denied1 ⊆ *setbydecision-all iface rs1 FinalDeny* ⟹
no-spoofing-algorithm iface ipassmt rs2 allowed denied1 ⟹
nospoof iface ipassmt (*rs1*@*rs2*)
proof(*induction iface ipassmt rs2 allowed denied1 arbitrary*: *rs1 allowed denied1*

rule: *no-spoofing-algorithm.induct*)
case (*1 iface ipassmt*)
from *1* **have** *allowed* $-$ *denied1* $\subseteq$ *ipv4cidr-union-set* (*set* (*the* (*ipassmt iface*)))
by(*simp*)
with *1* **have** *setbydecision iface rs1 FinalAllow* $-$ *setbydecision-all iface rs1 FinalDeny*
$\subseteq$ *ipv4cidr-union-set* (*set* (*the* (*ipassmt iface*)))
by *blast*

thus *?case*
by(*simp add*: *nospoof-setbydecision setbydecision-setbydecision-all-Allow*)
next
case (*2 iface ipassmt m rs*)
from *2*(*2*) **have** *simple-rs1*: *simple-ruleset rs1* **by**(*simp add*: *simple-ruleset-def*)
hence *simple-rs'*: *simple-ruleset* (*rs1* @ [*Rule m Accept*]) **by**(*simp add*: *simple-ruleset-def*)
from *2*(*3*) **have** *simple-rs*: *simple-ruleset rs* **by**(*simp add*: *simple-ruleset-def*)
with *2* **have** *IH*: $\bigwedge$*rs' allowed denied1*.
simple-ruleset rs' $\Longrightarrow$
setbydecision iface rs' FinalAllow $\subseteq$ *allowed* $\Longrightarrow$
denied1 $\subseteq$ *setbydecision-all iface rs' FinalDeny* $\Longrightarrow$
no-spoofing-algorithm iface ipassmt rs allowed denied1 $\Longrightarrow$ *nospoof iface ipassmt* (*rs'* @ *rs*)
by(*simp*)
from *2*(*5*) *simple-rs'* **have** *setbydecision iface* (*rs1* @ [*Rule m Accept*]) *FinalAllow* $\subseteq$
(*allowed* $\cup$ {*ip*. $\exists$ *p*. *matches* (*common-matcher*, *in-doubt-allow*) *m Accept* (*p*⦇*p-iiface* := *iface-sel iface*, *p-src* := *ip*⦈)})
apply(*simp add*: *setbydecision-append*)
apply(*simp add*: *helper1*)
by *blast*
with *get-exists-matching-src-ips-subset 2*(*4*) **have** *allowed*: *setbydecision iface* (*rs1* @ [*Rule m Accept*]) *FinalAllow* $\subseteq$ (*allowed* $\cup$ *get-exists-matching-src-ips iface m*)
by *fastforce*

from *2*(*6*) *setbydecision-all-appendAccept*[*OF simple-rs'*] **have** *denied1*: *denied1* $\subseteq$ *setbydecision-all iface* (*rs1* @ [*Rule m Accept*]) *FinalDeny* **by** *simp*

from *2*(*7*) **have** *no-spoofing-algorithm-prems*: *no-spoofing-algorithm iface ipassmt rs*
(*allowed* $\cup$ *get-exists-matching-src-ips iface m*) *denied1*
by(*simp*)

from *IH*[*OF simple-rs' allowed denied1 no-spoofing-algorithm-prems*] **have** *nospoof iface ipassmt* ((*rs1* @ [*Rule m Accept*]) @ *rs*) **by** *blast*
thus *?case* **by**(*simp*)
next
case (*3 iface ipassmt m rs*)
from *3*(*2*) **have** *simple-rs1*: *simple-ruleset rs1* **by**(*simp add*: *simple-ruleset-def*)

hence *simple-rs′*: *simple-ruleset* (*rs1* @ [*Rule m Drop*]) **by**(*simp add*: *simple-ruleset-def*)
from *3*(*3*) **have** *simple-rs*: *simple-ruleset rs* **by**(*simp add*: *simple-ruleset-def*)
with *3* **have** *IH*: $\bigwedge$*rs′ allowed denied1*.
simple-ruleset rs′ $\Longrightarrow$
setbydecision iface rs′ FinalAllow $\subseteq$ *allowed* $\Longrightarrow$
denied1 $\subseteq$ *setbydecision-all iface rs′ FinalDeny* $\Longrightarrow$
no-spoofing-algorithm iface ipassmt rs allowed denied1 $\Longrightarrow$ *nospoof iface ipassmt* (*rs′* @ *rs*)
by(*simp*)
from *3*(*5*) *simple-rs′* **have** *allowed*: *setbydecision iface* (*rs1* @ [*Rule m Drop*]) *FinalAllow* $\subseteq$ *allowed*
by(*simp add*: *setbydecision-append*)

have {*ip*. $\forall$ *p*. *matches* (*common-matcher*, *in-doubt-allow*) *m Drop* (*p*(|*p-iiface* := *iface-sel iface*, *p-src* := *ip*|))} $\subseteq$
setbydecision-all iface [*Rule m Drop*] *FinalDeny* **by**(*simp add*: *setbydecision-all-def*)
with *3*(*5*) **have** *setbydecision-all iface rs1 FinalDeny* $\cup$ ({*ip*. $\forall$ *p*. *matches* (*common-matcher*, *in-doubt-allow*) *m Drop* (*p*(|*p-iiface* := *iface-sel iface*, *p-src* := *ip*|))} $-$ *allowed*) $\subseteq$
setbydecision-all iface rs1 FinalDeny $\cup$ (*setbydecision-all iface* [*Rule m Drop*] *FinalDeny* $-$ *setbydecision iface rs1 FinalAllow*)
by *blast*
with *3*(*6*) *setbydecision-all-append-subset2*[*OF simple-rs′*, *of iface*] **have**
denied1 $\cup$ ({*ip*. $\forall$ *p*. *matches* (*common-matcher*, *in-doubt-allow*) *m Drop* (*p*(|*p-iiface* := *iface-sel iface*, *p-src* := *ip*|))} $-$ *allowed*) $\subseteq$
setbydecision-all iface (*rs1* @ [*Rule m Drop*]) *FinalDeny*
by *blast*
with *get-all-matching-src-ips* *3*(*4*) **have** *denied1*:
denied1 $\cup$ (*get-all-matching-src-ips iface m* $-$ *allowed*) $\subseteq$ *setbydecision-all iface* (*rs1* @ [*Rule m Drop*]) *FinalDeny*
by *force*

from *3*(*7*) **have** *no-spoofing-algorithm-prems*: *no-spoofing-algorithm iface ipassmt rs allowed*
(*denied1* $\cup$ (*get-all-matching-src-ips iface m* $-$ *allowed*))
apply(*simp*)
done

from *IH*[*OF simple-rs′ allowed denied1 no-spoofing-algorithm-prems*] **have** *nospoof iface ipassmt* ((*rs1* @ [*Rule m Drop*]) @ *rs*) **by** *blast*
thus *?case* **by**(*simp*)
next
case *4-1* **thus** *?case* **by**(*simp add*: *simple-ruleset-def*)
next
case *4-2* **thus** *?case* **by**(*simp add*: *simple-ruleset-def*)
next
case *4-3* **thus** *?case* **by**(*simp add*: *simple-ruleset-def*)
next

case *4-4* **thus** *?case* **by**(*simp add*: *simple-ruleset-def*)
next
case *4-5* **thus** *?case* **by**(*simp add*: *simple-ruleset-def*)
next
case *4-6* **thus** *?case* **by**(*simp add*: *simple-ruleset-def*)
qed

definition *no-spoofing-iface* :: *iface* ⇒ *ipassignment* ⇒ *common-primitive rule list* ⇒ *bool* **where**
no-spoofing-iface iface ipassmt rs ≡ *no-spoofing-algorithm iface ipassmt rs* {} {}

lemma[*code*]: *no-spoofing-iface iface ipassmt rs* =
no-spoofing-algorithm-executable iface ipassmt rs Empty-WordInterval Empty-WordInterval
by(*simp add*: *no-spoofing-iface-def no-spoofing-algorithm-executable*)

private corollary *no-spoofing-algorithm-sound*: *simple-ruleset rs* ⟹ ∀ *r*∈*set rs*. *normalized-nnf-match* (*get-match r*) ⟹
no-spoofing-iface iface ipassmt rs ⟹ *nospoof iface ipassmt rs*
unfolding *no-spoofing-iface-def*
apply(*rule no-spoofing-algorithm-sound-generalized*[*of* [] *rs iface* {} {}, *simplified*])
apply(*simp-all*)
apply(*simp add*: *simple-ruleset-def*)
apply(*simp add*: *setbydecision-def*)
done

The *nospoof* definition used throughout the proofs corresponds to checking *no-spoofing* for all interfaces

private lemma *nospoof*: *simple-ruleset rs* ⟹ (∀ *iface* ∈ *dom ipassmt*. *nospoof iface ipassmt rs*) ⟷ *no-spoofing ipassmt rs*
unfolding *nospoof-def no-spoofing-def*
apply(*drule simple-imp-good-ruleset*)
apply(*subst approximating-semantics-iff-fun-good-ruleset*)
apply(*simp-all*)
done

theorem *no-spoofing-iface*: *simple-ruleset rs* ⟹ ∀ *r*∈*set rs*. *normalized-nnf-match* (*get-match r*) ⟹
∀ *iface* ∈ *dom ipassmt*. *no-spoofing-iface iface ipassmt rs* ⟹ *no-spoofing ipassmt rs*
by(*auto dest*: *nospoof no-spoofing-algorithm-sound*)

lemma *no-spoofing-iface*
(*Iface ''eth0''*)
[*Iface ''eth0''* ↦ [(*ipv4addr-of-dotdecimal* (*192*,*168*,*0*,*0*), *24*)]]
[*Rule* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *24*)))

(*Match* (*IIface* (*Iface* ''*eth0*''))) *action.Accept*,
Rule MatchAny action.Drop] **by** *eval*
end

value(*code*) *no-spoofing-iface* (*Iface* ''*eth1.1011*'') ([*Iface* ''*eth1.1011*'' ↦ [(*ipv4addr-of-dotdecimal* (*131*,*159*,*14*,*0*), *24*)]]:: *ipassignment*)
[*Rule* (*MatchNot* (*Match* (*IIface* (*Iface* ''*eth1.1011+*'')))) *action.Accept*,
Rule (*MatchAnd* (*MatchNot* (*Match* (*Src* (*Ip4AddrNetmask* (*131*,*159*,*14*,*0*) *24*)))) (*Match* (*IIface* (*Iface* ''*eth1.101*'')))) *action.Drop*,
Rule MatchAny action.Accept]

value(*code*) *no-spoofing-iface* (*Iface* ''*eth1.1011*'') ([*Iface* ''*eth1.1011*'' ↦ [(*ipv4addr-of-dotdecimal* (*131*,*159*,*14*,*0*), *24*)]]:: *ipassignment*)
[*Rule* (*Match* (*Src* (*Ip4AddrNetmask* (*127*, *0*, *0*, *0*) *8*))) *Drop*]

end
theory *Ports-Normalize*
imports *Common-Primitive-Matcher*
Primitive-Normalization
begin

28.1 Normalizing ports

context
begin

private fun *ipt-ports-negation-type-normalize* :: *ipt-ports negation-type* ⇒ *ipt-ports*
where
ipt-ports-negation-type-normalize (*Pos ps*) = *ps* |
ipt-ports-negation-type-normalize (*Neg ps*) = *ports-invert ps*

private lemma *ipt-ports-negation-type-normalize* (*Neg* [(*0*,*65535*)]) = [] **by** *eval*

declare *ipt-ports-negation-type-normalize.simps*[*simp del*]

private lemma *ipt-ports-negation-type-normalize-correct*:
matches (*common-matcher*, α) (*negation-type-to-match-expr-f* (*Src-Ports*) *ps*) *a p* ⟷
matches (*common-matcher*, α) (*Match* (*Src-Ports* (*ipt-ports-negation-type-normalize* *ps*))) *a p*
matches (*common-matcher*, α) (*negation-type-to-match-expr-f* (*Dst-Ports*) *ps*) *a p* ⟷
matches (*common-matcher*, α) (*Match* (*Dst-Ports* (*ipt-ports-negation-type-normalize* *ps*))) *a p*
apply(*case-tac* [!] *ps*)
apply(*simp-all add*: *ipt-ports-negation-type-normalize.simps matches-case-ternaryvalue-tuple*

bunch-of-lemmata-about-matches bool-to-ternary-simps ports-invert split: ternaryvalue.split)
done

ipt-ports list ⇒ *ipt-ports*

private definition *ipt-ports-andlist-compress* :: (*'a::len word* × *'a::len word*) *list list* ⇒ (*'a::len word* × *'a::len word*) *list* **where**
ipt-ports-andlist-compress pss = *br2l* (*fold* (*λps accu.* (*wordinterval-intersection* (*l2br ps*) *accu*)) *pss wordinterval-UNIV*)

private lemma *ipt-ports-andlist-compress-correct*: *ports-to-set* (*ipt-ports-andlist-compress pss*) = $\bigcap$ *set* (*map ports-to-set pss*)
proof −
{ **fix** *accu*
have *ports-to-set* (*br2l* (*fold* (*λps accu.* (*wordinterval-intersection* (*l2br ps*) *accu*)) *pss accu*)) = ($\bigcap$ *set* (*map ports-to-set pss*)) ∩ (*ports-to-set* (*br2l accu*))
apply(*induction pss arbitrary*: *accu*)
apply(*simp-all add*: *ports-to-set-wordinterval l2br-br2l*)
by *fast*
}
from *this*[*of wordinterval-UNIV*] **show** *?thesis*
unfolding *ipt-ports-andlist-compress-def* **by**(*simp add*: *ports-to-set-wordinterval l2br-br2l*)
qed

definition *ipt-ports-compress* :: *ipt-ports negation-type list* ⇒ *ipt-ports* **where**
ipt-ports-compress pss = *ipt-ports-andlist-compress* (*map ipt-ports-negation-type-normalize pss*)

private lemma *ipt-ports-compress-src-correct*:
matches (*common-matcher*, α) (*alist-and* (*NegPos-map Src-Ports ms*)) *a p* ⟷ *matches* (*common-matcher*, α) (*Match* (*Src-Ports* (*ipt-ports-compress ms*))) *a p*
proof(*induction ms*)
case *Nil* **thus** *?case* **by**(*simp add*: *ipt-ports-compress-def bunch-of-lemmata-about-matches ipt-ports-andlist-compress-correct*)
next
case (*Cons m ms*)
thus *?case* (**is** *?goal*)
proof(*cases m*)
case *Pos* **thus** *?goal* **using** *Cons.IH*
by(*simp add*: *ipt-ports-compress-def ipt-ports-andlist-compress-correct bunch-of-lemmata-about-matches*
ternary-to-bool-bool-to-ternary ipt-ports-negation-type-normalize.simps)
next
case (*Neg a*)
thus *?goal* **using** *Cons.IH*

apply(*simp add*: *ipt-ports-compress-def ipt-ports-andlist-compress-correct bunch-of-lemmata-about-matches ternary-to-bool-bool-to-ternary*)
apply(*simp add*: *matches-case-ternaryvalue-tuple bool-to-ternary-simps ports-invert ipt-ports-negation-type-normalize.simps split*: *ternary-value.split*)
done
qed
qed
private lemma *ipt-ports-compress-dst-correct*:
matches (*common-matcher*, α) (*alist-and* (*NegPos-map Dst-Ports ms*)) *a p* $\longleftrightarrow$ *matches* (*common-matcher*, α) (*Match* (*Dst-Ports* (*ipt-ports-compress ms*))) *a p*
proof(*induction ms*)
case *Nil* **thus** *?case* **by**(*simp add*: *ipt-ports-compress-def bunch-of-lemmata-about-matches ipt-ports-andlist-compress-correct*)
next
case (*Cons m ms*)
thus *?case* (**is** *?goal*)
proof(*cases m*)
case *Pos* **thus** *?goal* **using** *Cons.IH*
by(*simp add*: *ipt-ports-compress-def ipt-ports-andlist-compress-correct bunch-of-lemmata-about-matches ternary-to-bool-bool-to-ternary ipt-ports-negation-type-normalize.simps*)
next
case (*Neg a*)
thus *?goal* **using** *Cons.IH*
apply(*simp add*: *ipt-ports-compress-def ipt-ports-andlist-compress-correct bunch-of-lemmata-about-matches ternary-to-bool-bool-to-ternary*)
apply(*simp add*: *matches-case-ternaryvalue-tuple bool-to-ternary-simps ports-invert ipt-ports-negation-type-normalize.simps split*: *ternaryvalue.split*)
done
qed
qed

private lemma *ipt-ports-compress-matches-set*: *matches* (*common-matcher*, α) (*Match* (*Src-Ports* (*ipt-ports-compress ips*))) *a p* $\longleftrightarrow$ *p-sport p* $\in \bigcap$ *set* (*map* (*ports-to-set* $\circ$ *ipt-ports-negation-type-normalize*) *ips*)
apply(*simp add*: *ipt-ports-compress-def*)
apply(*induction ips*)
apply(*simp*)
apply(*simp add*: *ipt-ports-compress-def bunch-of-lemmata-about-matches ipt-ports-andlist-compress-correct*)
apply(*rename-tac m ms*)
apply(*case-tac m*)
apply(*simp add*: *ipt-ports-andlist-compress-correct bunch-of-lemmata-about-matches ternary-to-bool-bool-to-ternary ipt-ports-negation-type-normalize.simps*)
apply(*simp add*: *ipt-ports-andlist-compress-correct bunch-of-lemmata-about-matches ternary-to-bool-bool-to-ternary*)

done

private lemma *singletonize-SrcDst-Ports*: *match-list* (*common-matcher*, α) (*map* (λ*spt*. (*MatchAnd* (*Match* (*Src-Ports* [*spt*]))) *ms*) (*spts*)) *a p* $\longleftrightarrow$
matches (*common-matcher*, α) (*MatchAnd* (*Match* (*Src-Ports spts*)) *ms*) *a p*
match-list (*common-matcher*, α) (*map* (λ*spt*. (*MatchAnd* (*Match* (*Dst-Ports* [*spt*]))) *ms*) (*dpts*)) *a p* $\longleftrightarrow$
matches (*common-matcher*, α) (*MatchAnd* (*Match* (*Dst-Ports dpts*)) *ms*) *a p*
apply(*simp-all add*: *match-list-matches bunch-of-lemmata-about-matches*(*1*) *multiports-disjuction*)
done

value *case primitive-extractor* (*is-Src-Ports*, *src-ports-sel*) *m*
of (*spts*, *rst*) $\Rightarrow$ *map* (λ*spt*. (*MatchAnd* (*Match* (*Src-Ports* [*spt*]))) *rst*) (*ipt-ports-compress spts*)

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-ports-step* :: ((*common-primitive* $\Rightarrow$ *bool*) $\times$ (*common-primitive* $\Rightarrow$ *ipt-ports*)) $\Rightarrow$
(*ipt-ports* $\Rightarrow$ *common-primitive*) $\Rightarrow$
common-primitive match-expr $\Rightarrow$ *common-primitive match-expr list* **where**
normalize-ports-step (*disc-sel*) *C* = *normalize-primitive-extract disc-sel C* (λ*me*. *map* (λ*pt*. [*pt*]) (*ipt-ports-compress me*))

definition *normalize-src-ports* :: *common-primitive match-expr* $\Rightarrow$ *common-primitive match-expr list* **where**
normalize-src-ports = *normalize-ports-step* (*is-Src-Ports*, *src-ports-sel*) *Src-Ports*

definition *normalize-dst-ports* :: *common-primitive match-expr* $\Rightarrow$ *common-primitive match-expr list* **where**
normalize-dst-ports = *normalize-ports-step* (*is-Dst-Ports*, *dst-ports-sel*) *Dst-Ports*

lemma *normalize-src-ports*: **assumes** *normalized-nnf-match m* **shows**
match-list (*common-matcher*, α) (*normalize-src-ports m*) *a p* $\longleftrightarrow$ *matches* (*common-matcher*, α) *m a p*
proof –
{ fix *ml*
have *match-list* (*common-matcher*, α) (*map* (*Match* $\circ$ *Src-Ports*) (*map* (λ*pt*. [*pt*]) (*ipt-ports-compress ml*))) *a p* =
matches (*common-matcher*, α) (*alist-and* (*NegPos-map Src-Ports ml*)) *a p*
by(*simp add*: *match-list-matches ipt-ports-compress-src-correct multiports-disjuction*)

} **with** *normalize-primitive-extract*[*OF assms wf-disc-sel-common-primitive*(*1*), **where** γ=(*common-matcher*, α)]
show *?thesis*
unfolding *normalize-src-ports-def normalize-ports-step-def* **by** *simp*
qed

lemma *normalize-dst-ports*: **assumes** *normalized-nnf-match m* **shows**
match-list (*common-matcher*, α) (*normalize-dst-ports m*) *a p* $\longleftrightarrow$ *matches* (*common-matcher*, α) *m a p*
proof –
{ **fix** *ml*
have *match-list* (*common-matcher*, α) (*map* (*Match* $\circ$ *Dst-Ports*) (*map* (λ*pt*. [*pt*]) (*ipt-ports-compress ml*))) *a p* =
matches (*common-matcher*, α) (*alist-and* (*NegPos-map Dst-Ports ml*)) *a p*
by(*simp add*: *match-list-matches ipt-ports-compress-dst-correct multiports-disjuction*)
} **with** *normalize-primitive-extract*[*OF assms wf-disc-sel-common-primitive*(*2*), **where** γ=(*common-matcher*, α)]
show *?thesis*
unfolding *normalize-dst-ports-def normalize-ports-step-def* **by** *simp*
qed

value *normalized-nnf-match* (*MatchAnd* (*MatchNot* (*Match* (*Src-Ports* [(*1*,*2*)]))) (*Match* (*Src-Ports* [(*1*,*2*)])))
value *normalize-src-ports* (*MatchAnd* (*MatchNot* (*Match* (*Src-Ports* [(*5*,*9*)]))) (*Match* (*Src-Ports* [(*1*,*2*)])))

value *normalize-src-ports* (*MatchAnd* (*MatchNot* (*Match* (*Prot* (*Proto TCP*)))) (*Match* (*Prot* (*ProtoAny*))))

fun *normalized-src-ports* :: *common-primitive match-expr* ⇒ *bool* **where**
normalized-src-ports MatchAny = *True* |
normalized-src-ports (*Match* (*Src-Ports* [])) = *True* |
normalized-src-ports (*Match* (*Src-Ports* [-])) = *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* :: *common-primitive match-expr* ⇒ *bool* **where**
normalized-dst-ports MatchAny = *True* |
normalized-dst-ports (*Match* (*Dst-Ports* [])) = *True* |
normalized-dst-ports (*Match* (*Dst-Ports* [-])) = *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* ∧ *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*pts*. *length pts* ≤ *1*) *ms*
by(*induction ms rule*: *normalized-src-ports.induct*, *simp-all*)
lemma *normalized-dst-ports-def2*: *normalized-dst-ports ms* = *normalized-n-primitive* (*is-Dst-Ports*, *dst-ports-sel*) (λ*pts*. *length pts* ≤ *1*) *ms*
by(*induction ms rule*: *normalized-dst-ports.induct*, *simp-all*)

private lemma *normalized-nnf-match-MatchNot-D*: *normalized-nnf-match* (*MatchNot m*) ⟹ *normalized-nnf-match m*
apply(*induction m*)
apply(*simp-all*)
done

private lemma ∀ *spt* ∈ *set* (*ipt-ports-compress spts*). *normalized-src-ports* (*Match* (*Src-Ports* [*spt*])) **by**(*simp*)

lemma *normalize-src-ports-normalized-n-primitive*: *normalized-nnf-match m* ⟹

∀ *m′* ∈ *set* (*normalize-src-ports m*). *normalized-src-ports m′*
unfolding *normalize-src-ports-def normalize-ports-step-def*
unfolding *normalized-src-ports-def2*
apply(*rule normalize-primitive-extract-normalizes-n-primitive*[*OF - wf-disc-sel-common-primitive*(*1*)])
by(*simp-all*)

lemma *normalized-nnf-match m* ⟹
∀ *m′* ∈ *set* (*normalize-src-ports m*). *normalized-src-ports m′* ∧ *normalized-nnf-match m′*
apply(*intro ballI*, *rename-tac mn*)
apply(*rule conjI*)
apply(*simp add*: *normalize-src-ports-normalized-n-primitive*)
unfolding *normalize-src-ports-def normalize-ports-step-def*

unfolding *normalized-dst-ports-def2*
by(*auto dest*: *normalize-primitive-extract-preserves-nnf-normalized*[*OF - wf-disc-sel-common-primitive*(*1*)])

lemma *normalize-dst-ports-normalized-n-primitive*: *normalized-nnf-match m* $\Longrightarrow$

$\forall m' \in$ *set* (*normalize-dst-ports m*). *normalized-dst-ports m'*
unfolding *normalize-dst-ports-def normalize-ports-step-def*
unfolding *normalized-dst-ports-def2*
apply(*rule normalize-primitive-extract-normalizes-n-primitive*[*OF - wf-disc-sel-common-primitive*(*2*)])
by(*simp-all*)

lemma *normalized-nnf-match m* $\Longrightarrow$ *normalized-dst-ports m* $\Longrightarrow$
$\forall mn \in$ *set* (*normalize-src-ports m*). *normalized-dst-ports mn*
unfolding *normalized-dst-ports-def2 normalize-src-ports-def normalize-ports-step-def*
apply(*frule*(*1*) *normalize-primitive-extract-preserves-unrelated-normalized-n-primitive*[*OF - - wf-disc-sel-common-primitive*(*1*), **where** *f*=(λ*me*. *map* (λ*pt*. [*pt*]) (*ipt-ports-compress me*))])
apply(*simp-all*)
done

end
end
theory *IpAddresses-Normalize*
imports *Common-Primitive-Matcher*
Primitive-Normalization
begin

28.2 Normalizing IP Addresses

fun *normalized-src-ips* :: *common-primitive match-expr* $\Rightarrow$ *bool* **where**
normalized-src-ips MatchAny = *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* $\wedge$ *normalized-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*) (λ*ip*. *True*) *ms*
by(*induction ms rule*: *normalized-src-ips.induct*, *simp-all*)

fun *normalized-dst-ips* :: *common-primitive match-expr* $\Rightarrow$ *bool* **where**
normalized-dst-ips MatchAny = *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* ∧ *normalized-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*) (λ*ip*. *True*) *ms*
by(*induction ms rule*: *normalized-dst-ips.induct*, *simp-all*)

value *normalize-primitive-extract* (*is-Src*, *src-sel*) *Src ipt-ipv4range-compress*
(*MatchAnd* (*MatchNot* (*Match* (*Src-Ports* [(*1*,*2*)]))) (*Match* (*Src-Ports* [(*1*,*2*)])))
value *normalize-primitive-extract* (*is-Src*, *src-sel*) *Src ipt-ipv4range-compress*
(*MatchAnd* (*MatchNot* (*Match* (*Src* (*Ip4AddrNetmask* (*10*,*0*,*0*,*0*) *2*)))) (*Match* (*Src-Ports* [(*1*,*2*)])))
value *normalize-primitive-extract* (*is-Src*, *src-sel*) *Src ipt-ipv4range-compress*
(*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*10*,*0*,*0*,*0*) *2*))) (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*10*,*0*,*0*,*0*) *8*))) (*Match* (*Src-Ports* [(*1*,*2*)]))))
value *normalize-primitive-extract* (*is-Src*, *src-sel*) *Src ipt-ipv4range-compress*
(*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*10*,*0*,*0*,*0*) *2*))) (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*192*,*0*,*0*,*0*) *8*))) (*Match* (*Src-Ports* [(*1*,*2*)]))))

definition *normalize-src-ips* :: *common-primitive match-expr* ⇒ *common-primitive match-expr list* **where**
normalize-src-ips = *normalize-primitive-extract* (*common-primitive.is-Src*, *src-sel*) *common-primitive.Src ipt-ipv4range-compress*

lemma *ipt-ipv4range-compress-src-matching*: *match-list* (*common-matcher*, α) (*map* (*Match* ∘ *Src*) (*ipt-ipv4range-compress ml*)) *a p* ⟷
matches (*common-matcher*, α) (*alist-and* (*NegPos-map Src ml*)) *a p*
proof −
have *matches* (*common-matcher*, α) (*alist-and* (*NegPos-map common-primitive.Src ml*)) *a p* ⟷
(∀ *m* ∈ *set* (*getPos ml*). *matches* (*common-matcher*, α) (*Match* (*Src m*)) *a p*) ∧
(∀ *m* ∈ *set* (*getNeg ml*). *matches* (*common-matcher*, α) (*MatchNot* (*Match* (*Src m*))) *a p*)
by(*induction ml rule*: *alist-and.induct*) (*auto simp add*: *bunch-of-lemmata-about-matches ternary-to-bool-bool-to-ternary*)
also have ... ⟷ *p-src p* ∈ (⋂ *ip* ∈ *set* (*getPos ml*). *ipv4s-to-set ip*) − (⋃ *ip* ∈ *set* (*getNeg ml*). *ipv4s-to-set ip*)
by(*simp add*: *match-simplematcher-SrcDst match-simplematcher-SrcDst-not*)
also have ... ⟷ *p-src p* ∈ (⋃ *ip* ∈ *set* (*ipt-ipv4range-compress ml*). *ipv4s-to-set ip*) **using** *ipt-ipv4range-compress* **by** *presburger*
also have ... ⟷ (∃ *ip* ∈ *set* (*ipt-ipv4range-compress ml*). *matches* (*common-matcher*, α) (*Match* (*Src ip*)) *a p*)

by(*simp add*: *match-simplematcher-SrcDst*)
finally show *?thesis* **using** *match-list-matches* **by** *fastforce*
qed
lemma *normalize-src-ips*: *normalized-nnf-match m* $\Longrightarrow$
match-list (*common-matcher*, α) (*normalize-src-ips m*) *a p* = *matches* (*common-matcher*, α) *m a p*
unfolding *normalize-src-ips-def*
using *normalize-primitive-extract*[*OF* - *wf-disc-sel-common-primitive*(*3*), **where** *f*=*ipt-ipv4range-compress* **and** γ=(*common-matcher*, α)]
ipt-ipv4range-compress-src-matching **by** *simp*

lemma *normalize-src-ips-normalized-n-primitive*: *normalized-nnf-match m* $\Longrightarrow$
$\forall$ *m'* $\in$ *set* (*normalize-src-ips m*). *normalized-src-ips m'*
unfolding *normalize-src-ips-def*
unfolding *normalized-src-ips-def2*
apply(*rule normalize-primitive-extract-normalizes-n-primitive*[*OF* - *wf-disc-sel-common-primitive*(*3*)])
by(*simp-all*)

definition *normalize-dst-ips* :: *common-primitive match-expr* $\Rightarrow$ *common-primitive match-expr list* **where**
normalize-dst-ips = *normalize-primitive-extract* (*common-primitive.is-Dst*, *dst-sel*) *common-primitive.Dst ipt-ipv4range-compress*

lemma *ipt-ipv4range-compress-dst-matching*: *match-list* (*common-matcher*, α) (*map* (*Match* $\circ$ *Dst*) (*ipt-ipv4range-compress ml*)) *a p* $\longleftrightarrow$
matches (*common-matcher*, α) (*alist-and* (*NegPos-map Dst ml*)) *a p*
proof −
have *matches* (*common-matcher*, α) (*alist-and* (*NegPos-map common-primitive.Dst ml*)) *a p* $\longleftrightarrow$
($\forall$ *m* $\in$ *set* (*getPos ml*). *matches* (*common-matcher*, α) (*Match* (*Dst m*)) *a p*) $\wedge$
($\forall$ *m* $\in$ *set* (*getNeg ml*). *matches* (*common-matcher*, α) (*MatchNot* (*Match* (*Dst m*))) *a p*)
by(*induction ml rule*: *alist-and.induct*) (*auto simp add*: *bunch-of-lemmata-about-matches ternary-to-bool-bool-to-ternary*)
also have ... $\longleftrightarrow$ *p-dst p* $\in$ ($\bigcap$ *ip* $\in$ *set* (*getPos ml*). *ipv4s-to-set ip*) − ($\bigcup$ *ip* $\in$ *set* (*getNeg ml*). *ipv4s-to-set ip*)
by(*simp add*: *match-simplematcher-SrcDst match-simplematcher-SrcDst-not*)
also have ... $\longleftrightarrow$ *p-dst p* $\in$ ($\bigcup$ *ip* $\in$ *set* (*ipt-ipv4range-compress ml*). *ipv4s-to-set ip*) **using** *ipt-ipv4range-compress* **by** *presburger*
also have ... $\longleftrightarrow$ ($\exists$ *ip* $\in$ *set* (*ipt-ipv4range-compress ml*). *matches* (*common-matcher*, α) (*Match* (*Dst ip*)) *a p*)
by(*simp add*: *match-simplematcher-SrcDst*)
finally show *?thesis* **using** *match-list-matches* **by** *fastforce*
qed
lemma *normalize-dst-ips*: *normalized-nnf-match m* $\Longrightarrow$
match-list (*common-matcher*, α) (*normalize-dst-ips m*) *a p* = *matches* (*common-matcher*, α) *m a p*

unfolding *normalize-dst-ips-def*
using *normalize-primitive-extract*[*OF* - *wf-disc-sel-common-primitive*(*4*), **where** *f* =*ipt-ipv4range-compress* **and** γ=(*common-matcher*, α)]
ipt-ipv4range-compress-dst-matching **by** *simp*

Normalizing the dst ips preserves the normalized src ips

lemma *normalized-nnf-match m* $\Longrightarrow$ *normalized-src-ips m* $\Longrightarrow$ $\forall$ *mn*$\in$*set* (*normalize-dst-ips m*). *normalized-src-ips mn*
unfolding *normalize-dst-ips-def normalized-src-ips-def2*
by(*rule normalize-primitive-extract-preserves-unrelated-normalized-n-primitive*) (*simp-all*)

lemma *normalize-dst-ips-normalized-n-primitive*: *normalized-nnf-match m* $\Longrightarrow$
$\forall$ *m'* $\in$ *set* (*normalize-dst-ips m*). *normalized-dst-ips m'*
unfolding *normalize-dst-ips-def normalized-dst-ips-def2*
by(*rule normalize-primitive-extract-normalizes-n-primitive*[*OF* - *wf-disc-sel-common-primitive*(*4*)]) (*simp-all*)

28.3 Inverting single network ranges

unused

fun *ipt-ipv4range-invert* :: *ipt-ipv4range* $\Rightarrow$ (*ipv4addr* $\times$ *nat*) *list* **where**
ipt-ipv4range-invert (*Ip4Addr addr*) = *ipv4range-split* (*wordinterval-invert* (*ipv4range-single* (*ipv4addr-of-dotdecimal addr*))) |
ipt-ipv4range-invert (*Ip4AddrNetmask base len*) = *ipv4range-split* (*wordinterval-invert*

(*prefix-to-range* (*ipv4addr-of-dotdecimal base AND NOT mask* (*32* − *len*), *len*)))

lemma *ipt-ipv4range-invert-case-Ip4Addr*: *ipt-ipv4range-invert* (*Ip4Addr addr*) = *ipt-ipv4range-invert* (*Ip4AddrNetmask addr 32*)
apply(*simp add*: *prefix-to-range-ipv4range-range pfxm-prefix-def ipv4range-single-def*)
apply(*subgoal-tac pfxm-mask* (*ipv4addr-of-dotdecimal addr*, *32*) = (*0*::*ipv4addr*))
apply(*simp add*: *ipv4range-range.simps*)
apply(*simp add*: *pfxm-mask-def pfxm-length-def*)
done

lemma *ipt-ipv4range-invert-case-Ip4AddrNetmask*:
($\bigcup$ ((λ (*base*, *len*). *ipv4range-set-from-bitmask base len*) ‘ (*set* (*ipt-ipv4range-invert* (*Ip4AddrNetmask base len*))))) =
− (*ipv4range-set-from-bitmask* (*ipv4addr-of-dotdecimal base*) *len*)
proof −

{ **fix** *r*
have $\forall$ *pfx*$\in$*set* (*ipv4range-split* (*wordinterval-invert r*)). *valid-prefix pfx* **using** *all-valid-Ball* **by** *blast*
with *prefix-bitrang-list-union* **have**
$\bigcup$((λ(*base*, *len*). *ipv4range-set-from-bitmask base len*) ' *set* (*ipv4range-split* (*wordinterval-invert r*))) =
wordinterval-to-set (*list-to-wordinterval* (*map prefix-to-range* (*ipv4range-split* (*wordinterval-invert r*)))) **by** *simp*
also have ... = *wordinterval-to-set* (*wordinterval-invert r*)
unfolding *wordinterval-eq-set-eq*[*symmetric*] **using** *ipv4range-split-union*[*of* (*wordinterval-invert r*)] *ipv4range-eq-def* **by** *simp*
also have ... = − *wordinterval-to-set r* **by** *auto*
finally have $\bigcup$((λ(*base*, *len*). *ipv4range-set-from-bitmask base len*) ' *set* (*ipv4range-split* (*wordinterval-invert r*))) = − *wordinterval-to-set r* **.**
} **from** *this*[*of* (*prefix-to-range* (*ipv4addr-of-dotdecimal base AND NOT mask* (*32* − *len*), *len*))]
show *?thesis*
apply(*simp only*: *ipt-ipv4range-invert.simps*)
apply(*simp add*: *prefix-to-range-set-eq*)
apply(*simp add*: *cornys-hacky-call-to-prefix-to-range-to-start-with-a-valid-prefix pfxm-length-def pfxm-prefix-def wordinterval-to-set-ipv4range-set-from-bitmask*)

by (*metis ipv4range-set-from-bitmask-alt1 ipv4range-set-from-netmask-base-mask-consume maskshift-eq-not-mask*)
qed

lemma *ipt-ipv4range-invert*: ($\bigcup$ ((λ (*base*, *len*). *ipv4range-set-from-bitmask base len*) ' (*set* (*ipt-ipv4range-invert ips*)))) = − *ipv4s-to-set ips*
apply(*cases ips*)
apply(*simp-all only*:)
prefer *2*
using *ipt-ipv4range-invert-case-Ip4AddrNetmask* **apply** *simp*
apply(*subst ipt-ipv4range-invert-case-Ip4Addr*)
apply(*subst ipt-ipv4range-invert-case-Ip4AddrNetmask*)
apply(*simp add*: *ipv4range-set-from-bitmask-32*)
done

lemma *matches* (*common-matcher*, α) (*MatchNot* (*Match* (*Src ip*))) *a p* ⟷ *p-src p* ∈ (− (*ipv4s-to-set ip*))
using *match-simplematcher-SrcDst-not* **by** *simp*
lemma *match-list-match-SrcDst*:
match-list (*common-matcher*, α) (*map* (*Match* ∘ *Src*) (*ips*::*ipt-ipv4range list*)) *a p* ⟷ *p-src p* ∈ ($\bigcup$ (*ipv4s-to-set* ' (*set ips*)))
match-list (*common-matcher*, α) (*map* (*Match* ∘ *Dst*) (*ips*::*ipt-ipv4range list*)) *a p* ⟷ *p-dst p* ∈ ($\bigcup$ (*ipv4s-to-set* ' (*set ips*)))
by(*simp-all add*: *match-list-matches match-simplematcher-SrcDst*)

lemma *match-list-ipt-ipv4range-invert*:

match-list (*common-matcher*, α) (*map* (*Match* $\circ$ *Src* $\circ$ (λ(*ip*, *n*). *Ip4AddrNetmask* (*dotdecimal-of-ipv4addr ip*) *n*)) (*ipt-ipv4range-invert ip*)) *a p* $\longleftrightarrow$
matches (*common-matcher*, α) (*MatchNot* (*Match* (*Src ip*))) *a p* (**is** *?m1* = *?m2*)
proof −
{**fix** *ips*
have *ipv4s-to-set* ' *set* (*map* (λ(*ip*, *n*). *Ip4AddrNetmask* (*dotdecimal-of-ipv4addr ip*) *n*) *ips*) =
(λ(*ip*, *n*). *ipv4range-set-from-bitmask ip n*) ' *set ips*
apply(*induction ips*)
apply(*simp*)
apply(*clarify*)
apply(*simp add*: *ipv4addr-of-dotdecimal-dotdecimal-of-ipv4addr*)
done
} **note** *myheper*=*this*[*of* (*ipt-ipv4range-invert ip*)]

from *match-list-match-SrcDst*[*of* - *map* (λ(*ip*, *n*). *Ip4AddrNetmask* (*dotdecimal-of-ipv4addr ip*) *n*) (*ipt-ipv4range-invert ip*)] **have**
?m1 = (*p-src p* $\in$ $\bigcup$(*ipv4s-to-set* ' *set* (*map* (λ(*ip*, *n*). *Ip4AddrNetmask* (*dotdecimal-of-ipv4addr ip*) *n*) (*ipt-ipv4range-invert ip*)))) **by** *simp*
also have ... = (*p-src p* $\in$ $\bigcup$((λ(*base*, *len*). *ipv4range-set-from-bitmask base len*) ' *set* (*ipt-ipv4range-invert ip*))) **using** *myheper* **by** *presburger*
also have ... = (*p-src p* $\in$ − *ipv4s-to-set ip*) **using** *ipt-ipv4range-invert*[*of ip*] **by** *simp*
also have ... = *?m2* **using** *match-simplematcher-SrcDst-not* **by** *simp*
finally show *?thesis* **.**
qed

lemma *matches* (*common-matcher*, α) (*match-list-to-match-expr*
(*map* (*Match* $\circ$ *Src* $\circ$ (λ(*ip*, *n*). *Ip4AddrNetmask* (*dotdecimal-of-ipv4addr ip*) *n*)) (*ipt-ipv4range-invert ip*))) *a p* $\longleftrightarrow$
matches (*common-matcher*, α) (*MatchNot* (*Match* (*Src ip*))) *a p*
apply(*subst match-list-ipt-ipv4range-invert*[*symmetric*])
apply(*simp add*: *match-list-to-match-expr-disjunction*)
done

end
theory *Transform*
imports *Common-Primitive-Matcher*
../Semantics-Ternary/Semantics-Ternary
../Semantics-Ternary/Negation-Type-Matching

../Primitive-Matchers/Ports-Normalize
../Primitive-Matchers/IpAddresses-Normalize
begin

definition *transform-optimize-dnf-strict* :: *common-primitive rule list* $\Rightarrow$ *common-primitive rule list* **where**
transform-optimize-dnf-strict = *optimize-matches opt-MatchAny-match-expr* $\circ$ *normalize-rules-dnf* $\circ$ (*optimize-matches* (*opt-MatchAny-match-expr* $\circ$ *optimize-primitive-univ*))

lemma *normalized-n-primitive-opt-MatchAny-match-expr*: *normalized-n-primitive disc-sel f m* $\Longrightarrow$ *normalized-n-primitive disc-sel f* (*opt-MatchAny-match-expr m*)
proof−
{ fix *disc*::($'a \Rightarrow$ *bool*) **and** *sel*::($'a \Rightarrow 'b$) **and** *n m1 m2*
have *normalized-n-primitive* (*disc*, *sel*) *n* (*opt-MatchAny-match-expr m1*) $\Longrightarrow$
normalized-n-primitive (*disc*, *sel*) *n* (*opt-MatchAny-match-expr m2*) $\Longrightarrow$
normalized-n-primitive (*disc*, *sel*) *n m1* $\wedge$ *normalized-n-primitive* (*disc*, *sel*) *n m2* $\Longrightarrow$
normalized-n-primitive (*disc*, *sel*) *n* (*opt-MatchAny-match-expr* (*MatchAnd m1 m2*))
by(*induction* (*MatchAnd m1 m2*) *rule*: *opt-MatchAny-match-expr.induct*) (*auto*)
}note *x=this*
assume *normalized-n-primitive disc-sel f m*
thus *?thesis*
apply(*induction disc-sel f m rule*: *normalized-n-primitive.induct*)
apply *simp-all*
using *x* **by** *simp*
qed

theorem *transform-optimize-dnf-strict*: **assumes** *simplers*: *simple-ruleset rs* **and** *wfα*: *wf-unknown-match-tac α*
shows (*common-matcher*, α),$p\vdash$ $\langle$*transform-optimize-dnf-strict rs*, *s*$\rangle \Rightarrow_\alpha t$ $\longleftrightarrow$ (*common-matcher*, α),$p\vdash$ $\langle$*rs*, *s*$\rangle \Rightarrow_\alpha t$
and *simple-ruleset* (*transform-optimize-dnf-strict rs*)
and $\forall$ *m* $\in$ *get-match* ‘ *set rs*. $\neg$ *has-disc C m* $\Longrightarrow$ $\forall$ *m* $\in$ *get-match* ‘ *set* (*transform-optimize-dnf-strict rs*). $\neg$ *has-disc C m*
and $\forall$ *m* $\in$ *get-match* ‘ *set* (*transform-optimize-dnf-strict rs*). *normalized-nnf-match m*
and $\forall$ *m* $\in$ *get-match* ‘ *set rs*. *normalized-n-primitive disc-sel f m* $\Longrightarrow$
$\forall$ *m* $\in$ *get-match* ‘ *set* (*transform-optimize-dnf-strict rs*). *normalized-n-primitive disc-sel f m*

proof −
let *?γ=(common-matcher, α)*
let *?fw=λrs. approximating-bigstep-fun ?γ p rs s*

have *simplers1*: *simple-ruleset (optimize-matches (opt-MatchAny-match-expr ∘ optimize-primitive-univ) rs)*
using *simplers optimize-matches-simple-ruleset* **by** (*metis*)

show *simplers-transform*: *simple-ruleset (transform-optimize-dnf-strict rs)*
unfolding *transform-optimize-dnf-strict-def*
using *simplers optimize-matches-simple-ruleset simple-ruleset-normalize-rules-dnf* **by** (*metis comp-apply*)

have *1*: $?\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t \longleftrightarrow$ *?fw rs = t*
using *approximating-semantics-iff-fun-good-ruleset[OF simple-imp-good-ruleset[OF simplers]]* **by** *fast*

have *?fw rs = ?fw (optimize-matches (opt-MatchAny-match-expr ∘ optimize-primitive-univ) rs)*
apply(*rule optimize-matches[symmetric]*)
using *optimize-primitive-univ-correct-matchexpr opt-MatchAny-match-expr-correct* **by** (*metis comp-apply*)
also have ... = *?fw (normalize-rules-dnf (optimize-matches (opt-MatchAny-match-expr ∘ optimize-primitive-univ) rs))*
apply(*rule normalize-rules-dnf-correct[symmetric]*)
using *simplers1* **by** (*metis good-imp-wf-ruleset simple-imp-good-ruleset*)
also have ... = *?fw (optimize-matches opt-MatchAny-match-expr (normalize-rules-dnf (optimize-matches (opt-MatchAny-match-expr ∘ optimize-primitive-univ) rs)))*
apply(*rule optimize-matches[symmetric]*)
using *opt-MatchAny-match-expr-correct* **by** (*metis*)
finally have *rs*: *?fw rs = ?fw (transform-optimize-dnf-strict rs)*
unfolding *transform-optimize-dnf-strict-def* **by** *auto*

have *2*: *?fw (transform-optimize-dnf-strict rs) = t* $\longleftrightarrow ?\gamma,p\vdash \langle$*transform-optimize-dnf-strict rs*$,\ s\rangle \Rightarrow_\alpha t$
using *approximating-semantics-iff-fun-good-ruleset[OF simple-imp-good-ruleset[OF simplers-transform], symmetric]* **by** *fast*
from *1 2 rs* **show** $?\gamma,p\vdash \langle$*transform-optimize-dnf-strict rs*$,\ s\rangle \Rightarrow_\alpha t \longleftrightarrow ?\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t$ **by** *simp*

have *tf1*: $\bigwedge$*r rs. transform-optimize-dnf-strict (r#rs) =*
(optimize-matches opt-MatchAny-match-expr (normalize-rules-dnf (optimize-matches (opt-MatchAny-match-expr ∘ optimize-primitive-univ) [r])))@
transform-optimize-dnf-strict rs
unfolding *transform-optimize-dnf-strict-def* **by**(*simp add: optimize-matches-def*)

— if the individual optimization functions preserve a property, then the whole

thing does
{ **fix** *P m*
assume *p1*: $\forall m.\ P\ m \longrightarrow P\ (\textit{optimize-primitive-univ}\ m)$
assume *p2*: $\forall m.\ P\ m \longrightarrow P\ (\textit{opt-MatchAny-match-expr}\ m)$
assume *p3*: $\forall m.\ P\ m \longrightarrow (\forall m' \in \textit{set}\ (\textit{normalize-match}\ m).\ P\ m')$
{ **fix** *rs*
have $\forall\ m \in \textit{get-match}\ `\ \textit{set}\ rs.\ P\ m \Longrightarrow \forall\ m \in \textit{get-match}\ `\ \textit{set}\ (\textit{optimize-matches}\ (\textit{opt-MatchAny-match-expr} \circ \textit{optimize-primitive-univ})\ rs).\ P\ m$
apply(*induction rs*)
apply(*simp add*: *optimize-matches-def*)
apply(*simp add*: *optimize-matches-def*)
using *p1 p2 p3* **by** *simp*
} **note** *opt1=this*
have $\forall\ m \in \textit{get-match}\ `\ \textit{set}\ rs.\ P\ m \Longrightarrow \forall\ m \in \textit{get-match}\ `\ \textit{set}\ (\textit{transform-optimize-dnf-strict}\ rs).\ P\ m$
apply(*drule opt1*)
apply(*induction rs*)
apply(*simp add*: *optimize-matches-def transform-optimize-dnf-strict-def*)
apply(*simp add*: *tf1 optimize-matches-def*)
apply(*safe*)
apply(*simp-all*)
using *p1 p2 p3* **by**(*simp*)
} **note** *matchpred-rule=this*

{ **fix** *m*
have $\neg\ \textit{has-disc}\ C\ m \Longrightarrow \neg\ \textit{has-disc}\ C\ (\textit{optimize-primitive-univ}\ m)$
by(*induction m rule*: *optimize-primitive-univ.induct*) *simp-all*
} **moreover** { **fix** *m*
have $\neg\ \textit{has-disc}\ C\ m \Longrightarrow \neg\ \textit{has-disc}\ C\ (\textit{opt-MatchAny-match-expr}\ m)$
by(*induction m rule*: *opt-MatchAny-match-expr.induct*) *simp-all*
} **moreover** { **fix** *m*
have $\neg\ \textit{has-disc}\ C\ m \longrightarrow (\forall m' \in \textit{set}\ (\textit{normalize-match}\ m).\ \neg\ \textit{has-disc}\ C\ m')$
by(*induction m rule*: *normalize-match.induct*) (*safe*,*auto*) — need safe, otherwise simplifier loops
} **ultimately show** $\forall\ m \in \textit{get-match}\ `\ \textit{set}\ rs.\ \neg\ \textit{has-disc}\ C\ m \Longrightarrow \forall\ m \in \textit{get-match}\ `\ \textit{set}\ (\textit{transform-optimize-dnf-strict}\ rs).\ \neg\ \textit{has-disc}\ C\ m$
using *matchpred-rule*[*of* $\lambda m.\ \neg\ \textit{has-disc}\ C\ m$] **by** *fast*

{ **fix** *P a*
have $(\textit{optimize-primitive-univ}\ (\textit{Match}\ a)) = (\textit{Match}\ a) \vee (\textit{optimize-primitive-univ}\ (\textit{Match}\ a)) = \textit{MatchAny}$
by(*induction* (*Match a*) *rule*: *optimize-primitive-univ.induct*) (*auto*)
hence $((\textit{optimize-primitive-univ}\ (\textit{Match}\ a)) = \textit{Match}\ a \Longrightarrow P\ a) \Longrightarrow (\textit{optimize-primitive-univ}\ (\textit{Match}\ a) = \textit{MatchAny} \Longrightarrow P\ a) \Longrightarrow P\ a$ **by** *blast*
} **note** *optimize-primitive-univ-match-cases=this*

{ **fix** *m*
have $\textit{normalized-n-primitive}\ \textit{disc-sel}\ f\ m \Longrightarrow \textit{normalized-n-primitive}\ \textit{disc-sel}\ f\ (\textit{optimize-primitive-univ}\ m)$

apply(*induction disc-sel f m rule*: *normalized-n-primitive.induct*)
apply(*simp-all split*: *split-if-asm*)
apply(*rule optimize-primitive-univ-match-cases*, *simp-all*)+
done
} **moreover** { **fix** *m*
have *normalized-n-primitive disc-sel f m* $\longrightarrow$ ($\forall$ *m'* $\in$ *set* (*normalize-match m*). *normalized-n-primitive disc-sel f m'*)
apply(*induction m rule*: *normalize-match.induct*)
apply(*simp-all*)[*2*]

apply(*case-tac disc-sel*) — no idea why the simplifier loops and this stuff and stuff and shit
apply(*clarify*)
apply(*simp*)
apply(*clarify*)
apply(*simp*)

apply(*safe*)
apply(*simp-all*)
done
} **ultimately show** $\forall$ *m* $\in$ *get-match* ' *set rs*. *normalized-n-primitive disc-sel f m* $\Longrightarrow$
$\forall$ *m* $\in$ *get-match* ' *set* (*transform-optimize-dnf-strict rs*). *normalized-n-primitive disc-sel f m*
using *matchpred-rule*[*of* λm. *normalized-n-primitive disc-sel f m*] *normalized-n-primitive-opt-MatchAny-m*
by *fast*

{ **fix** *rs*::*common-primitive rule list*
{ **fix** *m*::*common-primitive match-expr*
have *normalized-nnf-match m* $\Longrightarrow$ *normalized-nnf-match* (*opt-MatchAny-match-expr m*)
by(*induction m rule*: *opt-MatchAny-match-expr.induct*) (*simp-all*)
} **note** *x*=*this*
from *normalize-rules-dnf-normalized-nnf-match*[*of rs*]
have $\forall$ *x* $\in$ *set* (*normalize-rules-dnf rs*). *normalized-nnf-match* (*get-match x*)
.
hence $\forall$ *x* $\in$ *set* (*optimize-matches opt-MatchAny-match-expr* (*normalize-rules-dnf rs*)). *normalized-nnf-match* (*get-match x*)
apply(*induction rs rule*: *normalize-rules-dnf.induct*)
apply(*simp-all add*: *optimize-matches-def x*)
using *x* **by** *fastforce*
}
thus $\forall$ *m* $\in$ *get-match* ' *set* (*transform-optimize-dnf-strict rs*). *normalized-nnf-match m*
unfolding *transform-optimize-dnf-strict-def* **by** *simp*

qed

lemma *has-unknowns-common-matcher*: *has-unknowns common-matcher m* $\longleftrightarrow$ *has-disc is-Extra m*
proof –
{ **fix** *A p*
have *common-matcher A p* = *TernaryUnknown* $\longleftrightarrow$ *is-Extra A*
by(*induction A p rule*: *common-matcher.induct*) (*simp-all add*: *bool-to-ternary-Unknown*)
} **thus** *?thesis*
by(*induction common-matcher m rule*: *has-unknowns.induct*) (*simp-all*)
qed

definition *transform-remove-unknowns-generic* :: ($'a$, $'packet$) *match-tac* $\Rightarrow$ $'a$ *rule list* $\Rightarrow$ $'a$ *rule list* **where**
transform-remove-unknowns-generic γ = *optimize-matches-a* (*remove-unknowns-generic* γ)
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
shows (*common-matcher*, α),$p\vdash$ $\langle$*transform-remove-unknowns-generic* (*common-matcher*, α) *rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t* $\longleftrightarrow$ (*common-matcher*, α),$p\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t*
and *simple-ruleset* (*transform-remove-unknowns-generic* (*common-matcher*, α) *rs*)
and $\forall$ *m* $\in$ *get-match* ‘ *set rs*. $\neg$ *has-disc C m* $\Longrightarrow$
$\forall$ *m* $\in$ *get-match* ‘ *set* (*transform-remove-unknowns-generic* (*common-matcher*, α) *rs*). $\neg$ *has-disc C m*
and $\forall$ *m* $\in$ *get-match* ‘ *set* (*transform-remove-unknowns-generic* (*common-matcher*, α) *rs*). $\neg$ *has-unknowns common-matcher m*

and $\forall$ *m* $\in$ *get-match* ‘ *set rs*. *normalized-n-primitive disc-sel f m* $\Longrightarrow$
$\forall$ *m* $\in$ *get-match* ‘ *set* (*transform-remove-unknowns-generic* (*common-matcher*, α) *rs*). *normalized-n-primitive disc-sel f m*
proof –
let *?*γ=(*common-matcher*, α)
let *?fw*=λ*rs*. *approximating-bigstep-fun ?*γ *p rs s*

show *simplers1*: *simple-ruleset* (*transform-remove-unknowns-generic ?*γ *rs*)
unfolding *transform-remove-unknowns-generic-def*
using *simplers optimize-matches-a-simple-ruleset* **by** *blast*

show *?*γ,$p\vdash$ $\langle$*transform-remove-unknowns-generic ?*γ *rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t* $\longleftrightarrow$ *?*γ,$p\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t*
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers1*]]
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers*]]
unfolding *transform-remove-unknowns-generic-def*

using *optimize-matches-a-simplers*[*OF simplers*] *remove-unknowns-generic* **by** *metis*

{ fix *a m*
have ¬ *has-disc C m* $\Longrightarrow$ ¬ *has-disc C* (*remove-unknowns-generic ?*γ *a m*)
by(*induction ?*γ *a m rule*: *remove-unknowns-generic.induct*) *simp-all*
} thus $\forall$ *m* $\in$ *get-match* ' *set rs*. ¬ *has-disc C m* $\Longrightarrow$
$\forall$ *m* $\in$ *get-match* ' *set* (*transform-remove-unknowns-generic ?*γ *rs*). ¬ *has-disc C m*
unfolding *transform-remove-unknowns-generic-def*
by(*induction rs*) (*simp-all add*: *optimize-matches-a-def*)

{ fix *a m*
have *normalized-n-primitive disc-sel f m* $\Longrightarrow$
normalized-n-primitive disc-sel f (*remove-unknowns-generic ?*γ *a m*)
by(*induction ?*γ *a m rule*: *remove-unknowns-generic.induct*) (*simp-all*,*cases disc-sel*, *simp*)
} thus $\forall$ *m* $\in$ *get-match* ' *set rs*. *normalized-n-primitive disc-sel f m* $\Longrightarrow$
$\forall$ *m* $\in$ *get-match* ' *set* (*transform-remove-unknowns-generic ?*γ *rs*). *normalized-n-primitive disc-sel f m*
unfolding *transform-remove-unknowns-generic-def*
by(*induction rs*) (*simp-all add*: *optimize-matches-a-def*)

from *simplers* **show** $\forall$ *m* $\in$ *get-match* ' *set* (*transform-remove-unknowns-generic* (*common-matcher*, α) *rs*). ¬ *has-unknowns common-matcher m*
unfolding *transform-remove-unknowns-generic-def*
apply(*induction rs*)
apply(*simp add*: *optimize-matches-a-def*)
apply(*simp add*: *optimize-matches-a-def simple-ruleset-tail*)
apply(*rule remove-unknowns-generic-specification*[*OF - packet-independent-*α *packet-independent-*β*-unknown-common-matcher*])
apply(*simp add*: *simple-ruleset-def*)
done
qed

definition *transform-normalize-primitives* :: *common-primitive rule list* $\Rightarrow$ *common-primitive rule list* **where**
transform-normalize-primitives =
normalize-rules normalize-dst-ips $\circ$
normalize-rules normalize-src-ips $\circ$
normalize-rules normalize-dst-ports $\circ$
normalize-rules normalize-src-ports

lemma *normalize-rules-match-list-semantics-3*:
assumes $\forall\, m\ a.\ \textit{normalized-nnf-match}\ m \longrightarrow \textit{match-list}\ \gamma\ (f\ m)\ a\ p = \textit{matches}\ \gamma\ m\ a\ p$
and *simple-ruleset rs*
and *normalized*: $\forall\ m \in \textit{get-match}\ `\ \textit{set}\ rs.\ \textit{normalized-nnf-match}\ m$
shows $\textit{approximating-bigstep-fun}\ \gamma\ p\ (\textit{normalize-rules}\ f\ rs)\ s = \textit{approximating-bigstep-fun}\ \gamma\ p\ rs\ s$
apply(*rule normalize-rules-match-list-semantics-2*)
using *normalized assms*(*1*) **apply** *blast*
using *assms*(*2*) **by** *simp*

lemma *normalize-rules-primitive-extract-preserves-nnf-normalized*: $\forall\, m \in \textit{get-match}\ `\ \textit{set}\ rs.\ \textit{normalized-nnf-match}\ m \Longrightarrow \textit{wf-disc-sel}\ \textit{disc-sel}\ C \Longrightarrow$
$\forall\, m \in \textit{get-match}\ `\ \textit{set}\ (\textit{normalize-rules}\ (\textit{normalize-primitive-extract}\ \textit{disc-sel}\ C\ f)\ rs).\ \textit{normalized-nnf-match}\ m$
apply(*rule normalize-rules-preserves*[**where** P=*normalized-nnf-match* **and** f=(*normalize-primitive-extract disc-sel C f*)])
apply(*simp*)
apply(*cases disc-sel*)
using *normalize-primitive-extract-preserves-nnf-normalized* **by** *fast*

thm *normalize-primitive-extract-preserves-unrelated-normalized-n-primitive*
lemma *normalize-rules-preserves-unrelated-normalized-n-primitive*:
assumes $\forall\ m \in \textit{get-match}\ `\ \textit{set}\ rs.\ \textit{normalized-nnf-match}\ m \wedge \textit{normalized-n-primitive}\ (\textit{disc2}, \textit{sel2})\ P\ m$
and *wf-disc-sel* (*disc1*, *sel1*) *C*
and $\forall\, a.\ \neg\ \textit{disc2}\ (C\ a)$
shows $\forall\, m \in \textit{get-match}\ `\ \textit{set}\ (\textit{normalize-rules}\ (\textit{normalize-primitive-extract}\ (\textit{disc1}, \textit{sel1})\ C\ f)\ rs).\ \textit{normalized-nnf-match}\ m \wedge \textit{normalized-n-primitive}\ (\textit{disc2}, \textit{sel2})\ P\ m$
thm *normalize-rules-preserves*[**where** $P{=}\lambda m.\ \textit{normalized-nnf-match}\ m \wedge \textit{normalized-n-primitive}\ (\textit{disc2}, \textit{sel2})\ P\ m$
and f=*normalize-primitive-extract* (*disc1*, *sel1*) *C f*]
apply(*rule normalize-rules-preserves*[**where** $P{=}\lambda m.\ \textit{normalized-nnf-match}\ m \wedge \textit{normalized-n-primitive}\ (\textit{disc2}, \textit{sel2})\ P\ m$
and f=*normalize-primitive-extract* (*disc1*, *sel1*) *C f*])
using *assms*(*1*) **apply**(*simp*)
apply(*safe*)
using *normalize-primitive-extract-preserves-nnf-normalized*[*OF - assms*(*2*)] **apply** *fast*
using *normalize-primitive-extract-preserves-unrelated-normalized-n-primitive*[*OF - - assms*(*2*) *assms*(*3*)] **by** *blast*

lemma *normalize-rules-normalized-n-primitive*:
assumes $\forall\ m \in$ *get-match* ‘ *set rs*. *normalized-nnf-match m*
and $\forall\, m$. *normalized-nnf-match m* $\longrightarrow$
($\forall\, m' \in$ *set* (*normalize-primitive-extract* (*disc*, *sel*) *C f m*). *normalized-n-primitive* (*disc*, *sel*) *P m'*)
shows $\forall\, m \in$ *get-match* ‘ *set* (*normalize-rules* (*normalize-primitive-extract* (*disc*, *sel*) *C f*) *rs*).
normalized-n-primitive (*disc*, *sel*) *P m*
apply(*rule normalize-rules-property*[**where** *P*=*normalized-nnf-match* **and** *f*=*normalize-primitive-extract* (*disc*, *sel*) *C f*])
using *assms*(*1*) **apply** *simp*
using *assms*(*2*) **by** *simp*

theorem *transform-normalize-primitives*:
assumes *simplers*: *simple-ruleset rs*
and *wf* α: *wf-unknown-match-tac* α
and *normalized*: $\forall\ m \in$ *get-match* ‘ *set rs*. *normalized-nnf-match m*
shows (*common-matcher*, α),$p\vdash$ $\langle$*transform-normalize-primitives rs*, *s*$\rangle \Rightarrow_\alpha t$ $\longleftrightarrow$ (*common-matcher*, α),$p\vdash$ $\langle$*rs*, *s*$\rangle \Rightarrow_\alpha t$
and *simple-ruleset* (*transform-normalize-primitives rs*)

and $\forall\, a$. $\neg$ *disc1* (*Src-Ports a*) $\Longrightarrow \forall\, a$. $\neg$ *disc1* (*Dst-Ports a*) $\Longrightarrow$
$\forall\, a$. $\neg$ *disc1* (*Src a*) $\Longrightarrow \forall\, a$. $\neg$ *disc1* (*Dst a*) $\Longrightarrow$
$\forall\ m \in$ *get-match* ‘ *set rs*. $\neg$ *has-disc disc1 m* $\Longrightarrow \forall\ m \in$ *get-match* ‘ *set* (*transform-normalize-primitives rs*). $\neg$ *has-disc disc1 m*
and $\forall\ m \in$ *get-match* ‘ *set* (*transform-normalize-primitives rs*). *normalized-nnf-match m*
and $\forall\ m \in$ *get-match* ‘ *set* (*transform-normalize-primitives rs*).
normalized-src-ports m $\wedge$ *normalized-dst-ports m* $\wedge$ *normalized-src-ips m* $\wedge$ *normalized-dst-ips m*
and $\forall\, a$. $\neg$ *disc2* (*Src-Ports a*) $\Longrightarrow \forall\, a$. $\neg$ *disc2* (*Dst-Ports a*) $\Longrightarrow \forall\, a$. $\neg$ *disc2* (*Src a*) $\Longrightarrow \forall\, a$. $\neg$ *disc2* (*Dst a*) $\Longrightarrow$
$\forall\ m \in$ *get-match* ‘ *set rs*. *normalized-n-primitive* (*disc2*, *sel2*) *f m* $\Longrightarrow$
$\forall\ m \in$ *get-match* ‘ *set* (*transform-normalize-primitives rs*). *normalized-n-primitive* (*disc2*, *sel2*) *f m*
proof –
let *?*γ=(*common-matcher*, α)
let *?fw*=λ*rs*. *approximating-bigstep-fun ?*γ *p rs s*

show *simplers-t*: *simple-ruleset* (*transform-normalize-primitives rs*)
unfolding *transform-normalize-primitives-def*
by(*simp add*: *simple-ruleset-normalize-rules simplers*)

let *?rs1*=*normalize-rules normalize-src-ports rs*
let *?rs2*=*normalize-rules normalize-dst-ports ?rs1*
let *?rs3*=*normalize-rules normalize-src-ips ?rs2*
let *?rs4*=*normalize-rules normalize-dst-ips ?rs3*

from *normalize-rules-primitive-extract-preserves-nnf-normalized*[*OF normalized wf-disc-sel-common-primitive*(*1*)]
normalize-src-ports-def normalize-ports-step-def
have *normalized-rs1*: $\forall$ *m* $\in$ *get-match* ' *set ?rs1*. *normalized-nnf-match m* **by** *presburger*
from *normalize-rules-primitive-extract-preserves-nnf-normalized*[*OF this wf-disc-sel-common-primitive*(*2*)]
normalize-dst-ports-def normalize-ports-step-def
have *normalized-rs2*: $\forall$ *m* $\in$ *get-match* ' *set ?rs2*. *normalized-nnf-match m* **by** *presburger*
from *normalize-rules-primitive-extract-preserves-nnf-normalized*[*OF this wf-disc-sel-common-primitive*(*3*)]
normalize-src-ips-def
have *normalized-rs3*: $\forall$ *m* $\in$ *get-match* ' *set ?rs3*. *normalized-nnf-match m* **by** *presburger*
from *normalize-rules-primitive-extract-preserves-nnf-normalized*[*OF this wf-disc-sel-common-primitive*(*4*)]
normalize-dst-ips-def
have *normalized-rs4*: $\forall$ *m* $\in$ *get-match* ' *set ?rs4*. *normalized-nnf-match m* **by** *presburger*
thus $\forall$ *m* $\in$ *get-match* ' *set* (*transform-normalize-primitives rs*). *normalized-nnf-match m*
unfolding *transform-normalize-primitives-def* **by** *simp*

show *?*γ,*p*$\vdash$ $\langle$*transform-normalize-primitives rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t* $\longleftrightarrow$ *?*γ,*p*$\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t*
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers-t*]]
unfolding *approximating-semantics-iff-fun-good-ruleset*[*OF simple-imp-good-ruleset*[*OF simplers*]]
unfolding *transform-normalize-primitives-def*
apply(*simp*)
apply(*subst normalize-rules-match-list-semantics-3*)
using *normalize-dst-ips* **apply** *simp*
using *simplers simple-ruleset-normalize-rules* **apply** *blast*
using *normalized-rs3* **apply** *simp*
apply(*subst normalize-rules-match-list-semantics-3*)
using *normalize-src-ips* **apply** *simp*
using *simplers simple-ruleset-normalize-rules* **apply** *blast*
using *normalized-rs2* **apply** *simp*
apply(*subst normalize-rules-match-list-semantics-3*)
using *normalize-dst-ports* **apply** *simp*
using *simplers simple-ruleset-normalize-rules* **apply** *blast*
using *normalized-rs1* **apply** *simp*
apply(*subst normalize-rules-match-list-semantics-3*)
using *normalize-src-ports* **apply** *simp*
using *simplers simple-ruleset-normalize-rules* **apply** *blast*
using *normalized* **apply** *simp*
by *simp*

from *normalize-src-ports-normalized-n-primitive*

have *normalized-src-ports*: ∀ *m* ∈ *get-match* ‘ *set ?rs1*. *normalized-src-ports m*
using *normalize-rules-property*[*OF normalized*, **where** *f*=*normalize-src-ports*
and *Q*=*normalized-src-ports*] **by** *fast*

from *normalize-dst-ports-normalized-n-primitive*
normalize-rules-property[*OF normalized-rs1*, **where** *f*=*normalize-dst-ports*
and *Q*=*normalized-dst-ports*]
have *normalized-dst-ports*: ∀ *m* ∈ *get-match* ‘ *set ?rs2*. *normalized-dst-ports m*
by *fast*
from *normalize-src-ips-normalized-n-primitive*
normalize-rules-property[*OF normalized-rs2*, **where** *f*=*normalize-src-ips*
and *Q*=*normalized-src-ips*]
have *normalized-src-ips*: ∀ *m* ∈ *get-match* ‘ *set ?rs3*. *normalized-src-ips m* **by**
fast
from *normalize-dst-ips-normalized-n-primitive*
normalize-rules-property[*OF normalized-rs3*, **where** *f*=*normalize-dst-ips*
and *Q*=*normalized-dst-ips*]
have *normalized-dst-ips*: ∀ *m* ∈ *get-match* ‘ *set ?rs4*. *normalized-dst-ips m* **by**
fast

from *normalize-rules-preserves-unrelated-normalized-n-primitive*[*of - is-Src-Ports*
src-ports-sel (λ*pts*. *length pts* ≤ *1*),
folded normalized-src-ports-def2 normalize-ports-step-def]
have *preserve-normalized-src-ports*: ⋀*rs disc sel C f*.
∀ *m*∈*get-match* ‘ *set rs*. *normalized-nnf-match m* ⟹
∀ *m*∈*get-match* ‘ *set rs*. *normalized-src-ports m* ⟹
wf-disc-sel (*disc*, *sel*) *C* ⟹
∀ *a*. ¬ *is-Src-Ports* (*C a*) ⟹
∀ *m*∈*get-match* ‘ *set* (*normalize-rules* (*normalize-primitive-extract* (*disc*, *sel*)
C f) *rs*). *normalized-src-ports m*
by *metis*
from *preserve-normalized-src-ports*[*OF normalized-rs1 normalized-src-ports wf-disc-sel-common-primitive*(2
where *f*=(λ*me*. *map* (λ*pt*. [*pt*]) (*ipt-ports-compress me*)),
folded normalize-ports-step-def normalize-dst-ports-def]
have *normalized-src-ports-rs2*: ∀ *m* ∈ *get-match* ‘ *set ?rs2*. *normalized-src-ports*
m **by** *force*
from *preserve-normalized-src-ports*[*OF normalized-rs2 normalized-src-ports-rs2*
wf-disc-sel-common-primitive(*3*),
where *f*=*ipt-ipv4range-compress*, *folded normalize-src-ips-def*]
have *normalized-src-ports-rs3*: ∀ *m* ∈ *get-match* ‘ *set ?rs3*. *normalized-src-ports*
m **by** *force*
from *preserve-normalized-src-ports*[*OF normalized-rs3 normalized-src-ports-rs3*
wf-disc-sel-common-primitive(*4*),
where *f*=*ipt-ipv4range-compress*, *folded normalize-dst-ips-def*]
have *normalized-src-ports-rs4*: ∀ *m* ∈ *get-match* ‘ *set ?rs4*. *normalized-src-ports*
m **by** *force*

from *normalize-rules-preserves-unrelated-normalized-n-primitive*[*of - is-Dst-Ports*

dst-ports-sel (λ*pts. length pts* $\leq$ *1*),
folded normalized-dst-ports-def2 normalize-ports-step-def]
have *preserve-normalized-dst-ports*: $\bigwedge$*rs disc sel C f*.
$\forall$ *m*$\in$*get-match* ' *set rs. normalized-nnf-match m* $\Longrightarrow$
$\forall$ *m*$\in$*get-match* ' *set rs. normalized-dst-ports m* $\Longrightarrow$
wf-disc-sel (*disc*, *sel*) *C* $\Longrightarrow$
$\forall$ *a*. $\neg$ *is-Dst-Ports* (*C a*) $\Longrightarrow$
$\forall$ *m*$\in$*get-match* ' *set* (*normalize-rules* (*normalize-primitive-extract* (*disc*, *sel*) *C f*) *rs*). *normalized-dst-ports m*
by *metis*
from *preserve-normalized-dst-ports*[*OF normalized-rs2 normalized-dst-ports wf-disc-sel-common-primitive*(*3*
where *f1*=*ipt-ipv4range-compress*, *folded normalize-src-ips-def*]
have *normalized-dst-ports-rs3*: $\forall$ *m* $\in$ *get-match* ' *set ?rs3*. *normalized-dst-ports m* **by** *force*
from *preserve-normalized-dst-ports*[*OF normalized-rs3 normalized-dst-ports-rs3 wf-disc-sel-common-primitive*(*4*),
where *f1*=*ipt-ipv4range-compress*, *folded normalize-dst-ips-def*]
have *normalized-dst-ports-rs4*: $\forall$ *m* $\in$ *get-match* ' *set ?rs4*. *normalized-dst-ports m* **by** *force*

from *normalize-rules-preserves-unrelated-normalized-n-primitive*[*of ?rs3 is-Src src-sel* λ-. *True*,
OF - wf-disc-sel-common-primitive(*4*),
where *f*=*ipt-ipv4range-compress*, *folded normalize-dst-ips-def normalized-src-ips-def2*]
normalized-rs3 normalized-src-ips
have *normalized-src-rs4*: $\forall$ *m* $\in$ *get-match* ' *set ?rs4*. *normalized-src-ips m* **by** *force*
from *normalized-src-ports-rs4 normalized-dst-ports-rs4 normalized-src-rs4 normalized-dst-ips*
show $\forall$ *m* $\in$ *get-match* ' *set* (*transform-normalize-primitives rs*).
normalized-src-ports m $\wedge$ *normalized-dst-ports m* $\wedge$ *normalized-src-ips m* $\wedge$ *normalized-dst-ips m*
unfolding *transform-normalize-primitives-def* **by** *force*

show $\forall$ *a*. $\neg$ *disc2* (*Src-Ports a*) $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc2* (*Dst-Ports a*) $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc2* (*Src a*) $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc2* (*Dst a*) $\Longrightarrow$
$\forall$ *m* $\in$ *get-match* ' *set rs. normalized-n-primitive* (*disc2*, *sel2*) *f m* $\Longrightarrow$
$\forall$ *m* $\in$ *get-match* ' *set* (*transform-normalize-primitives rs*). *normalized-n-primitive* (*disc2*, *sel2*) *f m*
proof −
assume $\forall$ *m*$\in$*get-match* ' *set rs. normalized-n-primitive* (*disc2*, *sel2*) *f m*
with *normalized* **have** *a'*: $\forall$ *m*$\in$*get-match* ' *set rs. normalized-nnf-match m* $\wedge$ *normalized-n-primitive* (*disc2*, *sel2*) *f m* **by** *blast*

assume *a-Src-Ports*: $\forall$ *a*. $\neg$ *disc2* (*Src-Ports a*)
assume *a-Dst-Ports*: $\forall$ *a*. $\neg$ *disc2* (*Dst-Ports a*)
assume *a-Src*: $\forall$ *a*. $\neg$ *disc2* (*Src a*)
assume *a-Dst*: $\forall$ *a*. $\neg$ *disc2* (*Dst a*)

from *normalize-rules-preserves-unrelated-normalized-n-primitive*[*OF a′ wf-disc-sel-common-primitive*(*1*),
of (*λme. map* (*λpt.* [*pt*]) (*ipt-ports-compress me*)),
folded normalize-src-ports-def normalize-ports-step-def] *a-Src-Ports*
have ∀ *m*∈*get-match* ‘ *set ?rs1*. *normalized-n-primitive* (*disc2*, *sel2*) *f m* **by** *simp*
with *normalized-rs1 normalize-rules-preserves-unrelated-normalized-n-primitive*[*OF - wf-disc-sel-common-primitive*(*2*) *a-Dst-Ports*,
of ?rs1 sel2 f (*λme. map* (*λpt.* [*pt*]) (*ipt-ports-compress me*)),
folded normalize-dst-ports-def normalize-ports-step-def]
have ∀ *m*∈*get-match* ‘ *set ?rs2*. *normalized-n-primitive* (*disc2*, *sel2*) *f m* **by** *blast*
with *normalized-rs2 normalize-rules-preserves-unrelated-normalized-n-primitive*[*OF - wf-disc-sel-common-primitive*(*3*) *a-Src*,
of ?rs2 sel2 f ipt-ipv4range-compress,
folded normalize-src-ips-def]
have ∀ *m*∈*get-match* ‘ *set ?rs3*. *normalized-n-primitive* (*disc2*, *sel2*) *f m* **by** *blast*
with *normalized-rs3 normalize-rules-preserves-unrelated-normalized-n-primitive*[*OF - wf-disc-sel-common-primitive*(*4*) *a-Dst*,
of ?rs3 sel2 f ipt-ipv4range-compress,
folded normalize-dst-ips-def]
have ∀ *m*∈*get-match* ‘ *set ?rs4*. *normalized-n-primitive* (*disc2*, *sel2*) *f m* **by** *blast*
thus *?thesis*
unfolding *transform-normalize-primitives-def* **by** *simp*
qed

{ fix *m* **and** *m′* **and** *disc*::(*common-primitive* ⇒ *bool*) **and** *sel*::(*common-primitive* ⇒ *′x*) **and** *C′*:: (*′x* ⇒ *common-primitive*)
and *f′*::(*′x negation-type list* ⇒ *′x list*)
assume *am*: ¬ *has-disc disc1 m*
and *nm*: *normalized-nnf-match m*
and *am′*: *m′* ∈ *set* (*normalize-primitive-extract* (*disc*, *sel*) *C′ f′ m*)
and *wfdiscsel*: *wf-disc-sel* (*disc*,*sel*) *C′*

and *disc-different*: ∀ *a*. ¬ *disc1* (*C′ a*)

from *disc-different* **have** *af*: ∀ *spts*. (∀ *a* ∈ *Match* ‘ *C′* ‘ *set* (*f′ spts*). ¬ *has-disc disc1 a*)
by(*simp*)

obtain *as ms* **where** *asms*: *primitive-extractor* (*disc*, *sel*) *m* = (*as*, *ms*) **by** *fastforce*

from *am′ asms* **have** *m′* ∈ (*λspt. MatchAnd* (*Match* (*C′ spt*)) *ms*) ‘ *set* (*f′ as*)
unfolding *normalize-primitive-extract-def* **by**(*simp*)

hence *goalrule*:$\forall$ *spt* $\in$ *set* $(f'$ *as*$)$. $\neg$ *has-disc disc1* (*Match* (C' *spt*)) $\Longrightarrow$ $\neg$ *has-disc disc1 ms* $\Longrightarrow$ $\neg$ *has-disc disc1* m' **by** *fastforce*

from *am primitive-extractor-correct*(*4*)[*OF nm wfdiscsel asms*] **have** *1*: $\neg$ *has-disc disc1 ms* **by** *simp*
from *af* **have** *2*: $\forall$ *spt* $\in$ *set* $(f'$ *as*$)$. $\neg$ *has-disc disc1* (*Match* (C' *spt*)) **by** *simp*

from *goalrule*[*OF 2 1*] **have** $\neg$ *has-disc disc1* m' **.**
moreover from *nm* **have** *normalized-nnf-match* m' **by** (*metis am' normalize-primitive-extract-preserves-wfdiscsel*)
ultimately have $\neg$ *has-disc disc1* m' $\wedge$ *normalized-nnf-match* m' **by** *simp*
}
hence *x*: $\bigwedge$*disc sel* C' f'. *wf-disc-sel* (*disc*, *sel*) C' $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc1* (C' *a*) $\Longrightarrow$ $\forall$ *m*. *normalized-nnf-match m* $\wedge$ $\neg$ *has-disc disc1 m* $\longrightarrow$ ($\forall$ $m'$$\in$*set* (*normalize-primitive-extract* (*disc*, *sel*) C' f' *m*). *normalized-nnf-match* m' $\wedge$ $\neg$ *has-disc disc1* m')
by *blast*

have $\forall$ *a*. $\neg$ *disc1* (*Src-Ports a*) $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc1* (*Dst-Ports a*) $\Longrightarrow$
$\forall$ *a*. $\neg$ *disc1* (*Src a*) $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc1* (*Dst a*) $\Longrightarrow$
$\forall$ *m* $\in$ *get-match* ‘ *set rs*. $\neg$ *has-disc disc1 m* $\wedge$ *normalized-nnf-match m* $\Longrightarrow$
$\forall$ *m* $\in$ *get-match* ‘ *set* (*transform-normalize-primitives rs*). *normalized-nnf-match m* $\wedge$ $\neg$ *has-disc disc1 m*
unfolding *transform-normalize-primitives-def*
apply(*simp*)
apply(*rule normalize-rules-preserves'*)+
apply(*simp*)
using *x*[*OF wf-disc-sel-common-primitive*(*1*),
of (λ*me*. *map* (λ*pt*. [*pt*]) (*ipt-ports-compress me*)),*folded normalize-src-ports-def normalize-ports-step-def*] **apply** *blast*
using *x*[*OF wf-disc-sel-common-primitive*(*2*),
of (λ*me*. *map* (λ*pt*. [*pt*]) (*ipt-ports-compress me*)),*folded normalize-dst-ports-def normalize-ports-step-def*] **apply** *blast*
using *x*[*OF wf-disc-sel-common-primitive*(*3*), *of ipt-ipv4range-compress*,*folded normalize-src-ips-def*] **apply** *blast*
using *x*[*OF wf-disc-sel-common-primitive*(*4*), *of ipt-ipv4range-compress*,*folded normalize-dst-ips-def*] **apply** *blast*
done

thus $\forall$ *a*. $\neg$ *disc1* (*Src-Ports a*) $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc1* (*Dst-Ports a*) $\Longrightarrow$
$\forall$ *a*. $\neg$ *disc1* (*Src a*) $\Longrightarrow$ $\forall$ *a*. $\neg$ *disc1* (*Dst a*) $\Longrightarrow$
$\forall$ *m* $\in$ *get-match* ‘ *set rs*. $\neg$ *has-disc disc1 m* $\Longrightarrow$ $\forall$ *m* $\in$ *get-match* ‘ *set* (*transform-normalize-primitives rs*). $\neg$ *has-disc disc1 m*
using *normalized* **by** *blast*
qed

end
theory *Iface-Attic*
imports *String ../Common/Negation-Type*
begin

lemma *xxx*: $(\lambda s.\ i@s)$ ‘ $(UNIV::string\ set) = \{i@cs \mid cs.\ True\}$
by *auto*
lemma *xxx2*: $\{s@cs \mid s\ cs.\ P\ s\} = (\bigcup\ s \in \{s \mid s.\ P\ s\}.\ (\lambda cs.\ s@cs)$ ‘ $(UNIV::string\ set))$
by *auto*
lemma *xxx3*: $length\ i = n \Longrightarrow \{s@cs \mid s\ cs.\ length\ s = n \wedge s \neq (i::string)\} = \{s@cs \mid s\ cs.\ length\ s = n\} - \{s@cs \mid s\ cs.\ s = i\}$
thm *xxx2*$[of\ \lambda s::string.\ length\ s = n \wedge s \neq i]$
by *auto*

lemma *xxx3'*: $n \leq length\ i \Longrightarrow \{s\ @\ cs \mid s\ cs.\ length\ s = n \wedge s \neq take\ n\ (i::string)\} = \{s@cs \mid s\ cs.\ length\ s = n\} - \{s@cs \mid s\ cs.\ s = take\ n\ i\}$
apply(*subst xxx3*)
apply(*simp*)
by *blast*

lemma $-\ range\ (op\ @\ (butlast\ i)) = UNIV - (op\ @\ (butlast\ i))$ ‘ $UNIV$
by *fast*

lemma *notprefix*: $c \neq take\ (length\ c)\ i \longleftrightarrow (\forall\ cs.\ c@cs \neq i)$
apply(*safe*)
apply(*simp*)
by (*metis append-take-drop-id*)

definition *common-prefix* :: *string* ⇒ *string* ⇒ *bool* **where**
common-prefix i c ≡ *take* (*min* (*length c*) (*length i*)) *c* = *take* (*min* (*length c*) (*length i*)) *i*
lemma *common-prefix-alt*: *common-prefix i c* $\longleftrightarrow (\exists\ cs1\ cs2.\ i@cs1 = c@cs2)$
unfolding *common-prefix-def*
apply(*safe*)
apply (*metis append-take-drop-id min-def order-refl take-all*)
by (*metis min.commute notprefix order-refl take-all take-take*)
lemma *no-common-prefix*: ¬ *common-prefix i c* $\longleftrightarrow (\forall\ cs1\ cs2.\ i@cs1 \neq c@cs2)$
using *common-prefix-alt* **by** *presburger*
lemma *common-prefix-commute*: *common-prefix a b* ⟷ *common-prefix b a*
unfolding *common-prefix-alt* **by** *metis*
lemma *common-prefix-append-longer*: $length\ c \geq length\ i \Longrightarrow$ *common-prefix i c* ⟷ *common-prefix i* (*c*@*cs*)
by(*simp add*: *common-prefix-def min-def*)

lemma *xxxxx*: *length c* $\geq$ *length i* $\Longrightarrow$ ($\forall$ *csa*. (*i*::*string*) @ *csa* $\neq$ *c* @ *cs*) $\longleftrightarrow$ $\neg$ *common-prefix i c*
  apply(*rule*)
   prefer *2*
   apply (*simp add*: *no-common-prefix*)
  apply(*subst*(*asm*) *notprefix*[*symmetric*])
  apply(*cases* (*length c*) > (*length i*))
   apply(*simp add*: *min-def common-prefix-def*)
  apply(*simp add*: *min-def common-prefix-def*)
  done

lemma *no-prefix-set-split*: $\{c@cs \mid c\ cs.\ \neg$ *common-prefix* (*i*::*string*) $c\}$ =
    $\{c@cs \mid c\ cs.$ *length c* $\geq$ *length i* $\wedge$ *take* (*length i*) $(c@cs) \neq i\} \cup$
    $\{c@cs \mid c\ cs.$ *length c* $\leq$ *length i* $\wedge$ *take* (*length c*) $i \neq c\}$ (**is** *?A* = *?B1* $\cup$ *?B2*)
  proof –
    have *srule*: $\bigwedge$ *P Q*. *P* = *Q* $\Longrightarrow$ $\{c$ @ $cs \mid c\ cs.\ P\ c\ cs\} = \{c$ @ $cs \mid c\ cs.\ Q\ c\ cs\}$ **by** *simp*

    have *a*: *?A* = $\{c@cs \mid c\ cs.\ (\forall\ cs1\ cs2.\ i@cs1 \neq c@cs2)\}$
      using *no-common-prefix* **by** *presburger*

    have *b1*: *?B1* = $\{c@cs \mid c\ cs.$ *length c* $\geq$ *length i* $\wedge$ $\neg$ *common-prefix i c*$\}$
      by (*metis* (*full-types*) *notprefix xxxxx*)
    have *b2*: *?B2* = $\{c@cs \mid c\ cs.$ *length c* $\leq$ *length i* $\wedge$ $\neg$ *common-prefix i c*$\}$
      apply(*rule srule*)
      apply(*simp add*: *fun-eq-iff*)
      apply(*intro allI iffI*)
       apply(*simp-all*)
       apply(*elim conjE*)
       apply(*subst*(*asm*) *neq-commute*)
       apply(*subst*(*asm*) *notprefix*)
       apply(*drule xxxxx*[**where** *cs*=[]])
       apply(*simp add*: *common-prefix-commute*)
      apply(*elim conjE*)
      apply(*subst neq-commute*)
      apply(*subst notprefix*)
      apply(*drule xxxxx*[**where** *cs*=[]])
      apply(*simp add*: *common-prefix-commute*)
      done

    have *?A* $\subseteq$ *?B1* $\cup$ *?B2*
      apply(*subst b1*)
      apply(*subst b2*)
      apply(*rule*)
      apply(*simp*)
      apply(*elim exE conjE*)
      apply(*case-tac length x* $\leq$ *length i*)
       apply(*auto*)[*1*]

by (*metis nat-le-linear*)
have *?B1* ∪ *?B2* ⊆ *?A*
apply(*subst b1*)
apply(*subst b2*)
by *blast*
from ‹*?A* ⊆ *?B1* ∪ *?B2*› ‹*?B1* ∪ *?B2* ⊆ *?A*› **show** *?thesis* **by** *blast*
qed

lemma *other-char*: $a \neq (\textit{char-of-nat}\ (\textit{Suc}\ (\textit{nat-of-char}\ a)))$
apply(*cases a*)
apply(*simp add*: *nat-of-char-def char-of-nat-def*)
oops

thm *Set.full-SetCompr-eq*
lemma $\neg$ (*range f*) = $\{u.\ \forall x.\ u \neq f\ x\}$ **by** *blast*
lemma *all-empty-string-False*: $(\forall\ cs::string.\ cs \neq [])\longleftrightarrow$ *False* **by** *simp*

some *common-prefix* sets

lemma $\{c \mid c.\ \textit{common-prefix}\ i\ c\} \subseteq \{c@cs \mid c\ cs.\ \textit{common-prefix}\ i\ c\}$
apply(*safe*)
apply(*simp add*: *common-prefix-alt*)
apply (*metis append-Nil*)
done
lemma $\{c@cs \mid c\ cs.\ \textit{length}\ i \leq \textit{length}\ c \wedge \textit{common-prefix}\ i\ c\} \subseteq \{c \mid c.\ \textit{common-prefix}\ i\ c\}$
apply(*safe*)
apply(*rule-tac x=c@cs* **in** *exI*)
apply(*simp*)
apply(*subst common-prefix-append-longer*[*symmetric*])
apply(*simp-all*)
done
lemma $\{c \mid c.\ \textit{common-prefix}\ i\ c\} \subseteq \{c@cs \mid c\ cs.\ \textit{length}\ i \geq \textit{length}\ c \wedge \textit{common-prefix}\ i\ c\}$
apply(*safe*)
apply(*subst*(*asm*) *common-prefix-def*)
apply(*case-tac length c* ≤ *length i*)
apply(*simp-all add*: *min-def split*: *split-if-asm*)
apply(*rule-tac x=c* **in** *exI*)
apply(*simp*)
apply(*simp add*: *common-prefix-def min-def*)
apply(*rule-tac x=take* (*length i*) *c* **in** *exI*)
apply(*simp*)
apply(*simp add*: *common-prefix-def min-def*)
by (*metis notprefix*)

lemma $- \{c \mid c\ .\ \neg\ \textit{common-prefix}\ i\ c\} = \{c \mid c.\ \textit{common-prefix}\ i\ c\}$
apply(*safe*)

apply(*simp add*: *common-prefix-alt*)
done
lemma *inv-neg-commonprefixset*:$-$ $\{c@cs \mid c\ cs.\ \neg\ common\text{-}prefix\ i\ c\} = \{c \mid c.\ common\text{-}prefix\ i\ c\}$
apply(*safe*)
apply *blast*
apply(*simp add*: *common-prefix-alt*)
done
lemma $- \{c@cs \mid c\ cs.\ length\ c \leq length\ i \wedge \neg\ common\text{-}prefix\ i\ c\} \subseteq \{i@cs \mid cs.\ True\}$
apply(*safe*)
apply(*subst*(*asm*) *common-prefix-def*)
apply(*simp add*: *min-def*)
oops

lemma $- \{i@cs \mid cs.\ True\} = \{c@cs \mid c\ cs.\ \neg\ common\text{-}prefix\ i\ c\} \cup \{c \mid c.\ length\ c < length\ i\}$
apply(*rule*)
prefer *2*
apply(*safe*)[*1*]
apply(*simp add*: *no-common-prefix*)
apply(*simp add*: *no-common-prefix*)
apply(*simp*)
thm *Compl-anti-mono*[**where** $B=\{i\ @\ cs \mid cs.\ True\}$ **and** $A=- \{c\ @\ cs \mid c\ cs.\ length\ c \leq length\ i \wedge \neg\ common\text{-}prefix\ i\ c\}$, *simplified*]
apply(*rule Compl-anti-mono*[**where** $B=\{i\ @\ cs \mid cs.\ True\}$ **and** $A=- (\{c@cs \mid c\ cs.\ \neg\ common\text{-}prefix\ i\ c\} \cup \{c \mid c.\ length\ c < length\ i\})$, *simplified*])
apply(*safe*)
apply(*simp*)
apply(*case-tac* (*length i*) $\leq$ *length x*)
apply(*erule-tac* $x=x$ **in** *allE*, *simp*)
apply(*simp add*: *common-prefix-alt*)

apply (*metis append-eq-append-conv-if notprefix*)
apply(*simp*)
done

lemma *xxx4*: $\{s@cs \mid s\ cs.\ length\ s \leq length\ i - 1 \wedge s \neq take\ (length\ s)\ (i{::}string)\} =$
$(\bigcup\ n \in \{..\ length\ i - 1\}.\ \{s@cs \mid s\ cs.\ length\ s = n \wedge s \neq take\ (n)\ i\})$ (**is** $?A = ?B$)
proof $-$
have *a*: $?A = (\bigcup s{\in}\{s \mid s.\ length\ s \leq length\ i - 1 \wedge s \neq take\ (length\ s)\ i\}.\ range\ (op\ @\ s))$ (**is** $?A{=}?A'$)

by *blast*
have $\bigwedge n.\ \{s@cs \mid s\ cs.\ length\ s = n \wedge s \neq take\ (n)\ i\} = (\bigcup s{\in}\{s\ |s.\ length\ s = n \wedge s \neq take\ (n)\ i\}.\ range\ (op\ @\ s))$ **by** *auto*
hence *b*: $?B = (\bigcup\ n \in \{..\ length\ i - 1\}.\ (\bigcup s{\in}\{s\ |s.\ length\ s = n \wedge s \neq take\ (n)\ i\}.\ range\ (op\ @\ s)))$ (**is** $?B{=}?B'$) **by** *presburger*
{
fix $N{::}nat$ **and** $P{::}string \Rightarrow nat \Rightarrow bool$
have $(\bigcup s{\in}\{s\ |s.\ length\ s \leq N \wedge P\ s\ N\}.\ range\ (op\ @\ s)) = (\bigcup\ n \in \{..\ N\}.\ (\bigcup s{\in}\{s\ |s.\ length\ s = n \wedge P\ s\ N\}.\ range\ (op\ @\ s)))$
by *auto*
} **from** *this*[*of length* $i - 1$ $\lambda s\ n.\ s \neq take\ (length\ s)\ i$]
have $?A' = (\bigcup\ n{\leq}length\ i - 1.\ \bigcup s{\in}\{s\ |s.\ length\ s = n \wedge s \neq take\ (length\ s)\ i\}.\ range\ (op\ @\ s))$ **by** *simp*
also have $\ldots = ?B'$ **by** *blast*
with *a b* **show** *?thesis* **by** *blast*
qed

end
theory *Iface-Negation*
imports *String*
Iface-Attic
begin

29 Network Interfaces with Negation Support

*I don't think I can find a good way to support negated interfaces. Probably we should stick with simple interfaces for now. Reasons why negated interfaces are a problem: * Conjunction of Negated and not Negated interface cannot be expressed of one interface * Negated interfaces cannot (without additional assumption) be translated to non-negated interfaces. Example: Neg "eth++", when trying to translate this to a set of non-negated interfaces, it must be possible to have the interface 'eth+' in this set, where the final + is NOT treated as wildcard!* **datatype** *iface* = *Iface string negation-type*

definition *ifaceAny* :: *iface* **where**
ifaceAny $\equiv$ *Iface* (*Pos* $''+''$)
definition *IfaceFalse* :: *iface* **where**
IfaceFalse $\equiv$ *Iface* (*Neg* $''+''$)*If the interface name ends in a "+", then any interface which begins with this name will match. (man iptables)*
Here is how iptables handles this wildcard on my system. A packet for the loopback interface `lo` *is matched by the following expressions*

- *lo*
- *lo+*
- *l+*
- *+*

It is not matched by the following expressions

- *lo++*
- *lo+++*
- *lo1+*
- *lo1*

By the way: `Warning: weird characters in interface ' ' ('/' and ' ' are not allowed by the kernel).`

29.1 Helpers for the interface name (*string*)

argument 1: interface as in firewall rule - Wildcard support argument 2: interface a packet came from - No wildcard support

fun *internal-iface-name-match* :: *string* ⇒ *string* ⇒ *bool* **where**
internal-iface-name-match [] [] ⟷ *True* |
internal-iface-name-match (*i*#*is*) [] ⟷ (*i* = *CHR* ''+'' ∧ *is* = []) |
internal-iface-name-match [] (-#-) ⟷ *False* |
internal-iface-name-match (*i*#*is*) (*p-i*#*p-is*) ⟷ (*if* (*i* = *CHR* ''+'' ∧ *is* = []) *then True else* (
(*p-i* = *i*) ∧ *internal-iface-name-match is p-is*
))

fun *iface-name-is-wildcard* :: *string* ⇒ *bool* **where**
iface-name-is-wildcard [] ⟷ *False* |
iface-name-is-wildcard [*s*] ⟷ *s* = *CHR* ''+'' |
iface-name-is-wildcard (-#*ss*) ⟷ *iface-name-is-wildcard ss*

lemma *iface-name-is-wildcard-alt*: *iface-name-is-wildcard eth* ⟷ *eth* ≠ [] ∧ *last eth* = *CHR* ''+''
apply(*induction eth rule*: *iface-name-is-wildcard.induct*)
apply(*simp-all*)
done

lemma *iface-name-is-wildcard-alt'*: *iface-name-is-wildcard eth* ⟷ *eth* ≠ [] ∧ *hd* (*rev eth*) = *CHR* ''+''
apply(*simp add*: *iface-name-is-wildcard-alt*)
using *hd-rev* **by** *fastforce*

lemma *iface-name-is-wildcard-fst*: *iface-name-is-wildcard* (*i* # *is*) ⟹ *is* ≠ [] ⟹ *iface-name-is-wildcard is*
by(*simp add*: *iface-name-is-wildcard-alt*)

fun *internal-iface-name-to-set* :: *string* ⇒ *string set* **where**
internal-iface-name-to-set i = (*if* ¬ *iface-name-is-wildcard i*
then
{*i*}
else

$\{(butlast\ i)@cs \mid cs.\ True\})$

lemma $\{(butlast\ i)@cs \mid cs.\ True\} = (\lambda s.\ (butlast\ i)@s)$ ‘ $(UNIV::string\ set)$
by *fastforce*

lemma *internal-iface-name-to-set*: *internal-iface-name-match i p-iface* $\longleftrightarrow$ *p-iface* $\in$ *internal-iface-name-to-set i*
apply(*induction i p-iface rule*: *internal-iface-name-match.induct*)
apply(*simp-all*)
apply(*safe*)
apply(*simp-all add*: *iface-name-is-wildcard-fst*)
apply (*metis* (*full-types*) *iface-name-is-wildcard.simps*(*3*) *list.exhaust*)
by (*metis append-butlast-last-id*)

29.2 Matching

fun *match-iface* :: *iface* $\Rightarrow$ *string* $\Rightarrow$ *bool* **where**
match-iface (*Iface* (*Pos i*)) *p-iface* $\longleftrightarrow$ *internal-iface-name-match i p-iface* |
match-iface (*Iface* (*Neg i*)) *p-iface* $\longleftrightarrow$ $\neg$ *internal-iface-name-match i p-iface*

— Examples

lemma *match-iface* (*Iface* (*Pos* ''lo'')) ''lo''
match-iface (*Iface* (*Pos* ''lo+'')) ''lo''
match-iface (*Iface* (*Pos* ''l+'')) ''lo''
match-iface (*Iface* (*Pos* ''+'')) ''lo''
$\neg$ *match-iface* (*Iface* (*Pos* ''lo++'')) ''lo''
$\neg$ *match-iface* (*Iface* (*Pos* ''lo+++'')) ''lo''
$\neg$ *match-iface* (*Iface* (*Pos* ''lo1+'')) ''lo''
$\neg$ *match-iface* (*Iface* (*Pos* ''lo1'')) ''lo''
match-iface (*Iface* (*Pos* ''+'')) ''eth0''
$\neg$ *match-iface* (*Iface* (*Neg* ''+'')) ''eth0''
$\neg$ *match-iface* (*Iface* (*Neg* ''eth+'')) ''eth0''
match-iface (*Iface* (*Neg* ''lo+'')) ''eth0''
$\neg$ *match-iface* (*Iface* (*Neg* ''lo+'')) ''loX''
$\neg$ *match-iface* (*Iface* (*Pos* '''')) ''loX''
match-iface (*Iface* (*Neg* '''')) ''loX''
$\neg$ *match-iface* (*Iface* (*Pos* ''foobar+'')) ''foo'' **by** *eval+*

lemma *match-ifaceAny*: *match-iface ifaceAny i*
by(*cases i*, *simp-all add*: *ifaceAny-def*)

lemma *match-IfaceFalse*: $\neg$ *match-iface IfaceFalse i*
by(*cases i*, *simp-all add*: *IfaceFalse-def*)

— *match-iface* explained by the individual cases

lemma *match-iface-case-pos-nowildcard*: $\neg$ *iface-name-is-wildcard i* $\Longrightarrow$ *match-iface* (*Iface* (*Pos i*)) *p-i* $\longleftrightarrow$ $i = p\text{-}i$
apply(*simp*)
apply(*induction i p-i rule*: *internal-iface-name-match.induct*)
apply(*auto simp add*: *iface-name-is-wildcard-alt split*: *split-if-asm*)
done

lemma *match-iface-case-neg-nowildcard*: $\neg$ *iface-name-is-wildcard* $i \Longrightarrow$ *match-iface* (*Iface* (*Neg* i)) *p-i* $\longleftrightarrow i \neq$ *p-i*
apply(*simp*)
apply(*induction i p-i rule*: *internal-iface-name-match.induct*)
apply(*auto simp add*: *iface-name-is-wildcard-alt split*: *split-if-asm*)
done
lemma *match-iface-case-pos-wildcard-prefix*:
iface-name-is-wildcard $i \Longrightarrow$ *match-iface* (*Iface* (*Pos* i)) *p-i* $\longleftrightarrow$ *butlast* i = *take* (*length* i − *1*) *p-i*
apply(*simp*)
apply(*induction i p-i rule*: *internal-iface-name-match.induct*)
apply(*simp-all*)
apply(*simp add*: *iface-name-is-wildcard-alt split*: *split-if-asm*)
apply(*intro conjI*)
apply(*simp add*: *iface-name-is-wildcard-alt split*: *split-if-asm*)
apply(*intro impI*)
apply(*simp add*: *iface-name-is-wildcard-fst*)
by (*metis One-nat-def length-0-conv list.sel*(*1*) *list.sel*(*3*) *take-Cons'*)
lemma *match-iface-case-pos-wildcard-length*: *iface-name-is-wildcard* $i \Longrightarrow$ *match-iface* (*Iface* (*Pos* i)) *p-i* $\Longrightarrow$ *length p-i* $\geq$ (*length* i − *1*)
apply(*simp*)
apply(*induction i p-i rule*: *internal-iface-name-match.induct*)
apply(*simp-all*)
apply(*simp add*: *iface-name-is-wildcard-alt split*: *split-if-asm*)
done
corollary *match-iface-case-pos-wildcard*:
iface-name-is-wildcard $i \Longrightarrow$ *match-iface* (*Iface* (*Pos* i)) *p-i* $\longleftrightarrow$ *butlast* i = *take* (*length* i − *1*) *p-i* $\wedge$ *length p-i* $\geq$ (*length* i − *1*)
using *match-iface-case-pos-wildcard-length match-iface-case-pos-wildcard-prefix* **by** *blast*
lemma *match-iface-case-neg-wildcard-prefix*: *iface-name-is-wildcard* $i \Longrightarrow$ *match-iface* (*Iface* (*Neg* i)) *p-i* $\longleftrightarrow$ *butlast* $i \neq$ *take* (*length* i − *1*) *p-i*
apply(*simp*)
apply(*induction i p-i rule*: *internal-iface-name-match.induct*)
apply(*simp-all*)
apply(*simp add*: *iface-name-is-wildcard-alt split*: *split-if-asm*)
apply(*intro conjI*)
apply(*simp add*: *iface-name-is-wildcard-alt split*: *split-if-asm*)
apply(*simp add*: *iface-name-is-wildcard-fst*)
by (*metis One-nat-def length-0-conv list.sel*(*1*) *list.sel*(*3*) *take-Cons'*)

lemma *match-iface* (*Iface* (*Pos* i)) *p-iface* $\longleftrightarrow$ *p-iface* $\in$ *internal-iface-name-to-set* i
using *internal-iface-name-to-set* **by** *simp*
lemma *match-iface* (*Iface* (*Neg* i)) *p-iface* $\longleftrightarrow$ *p-iface* $\notin$ *internal-iface-name-to-set* i
using *internal-iface-name-to-set* **by** *simp*

lemma *match-iface* (*Iface* (*Neg i*)) *p-iface* $\longleftrightarrow$ *p-iface* $\in -$ (*internal-iface-name-to-set i*)
  using *internal-iface-name-to-set* **by** *simp*
— beware of handling of + as normal non-wildcard character!
lemma *match-iface* (*Iface* (*Neg* ''*eth++*'')) ''*eth*'' **by** *eval*

definition *internal-iface-name-wildcard-longest* :: *string* $\Rightarrow$ *string* $\Rightarrow$ *string option* **where**
  internal-iface-name-wildcard-longest i1 i2 = (
    if
      take (*min* (*length i1* $-$ *1*) (*length i2* $-$ *1*)) *i1* = *take* (*min* (*length i1* $-$ *1*) (*length i2* $-$ *1*)) *i2*
    then
      Some (*if length i1* $\leq$ *length i2 then i2 else i1*)
    else
      None)
lemma *internal-iface-name-wildcard-longest* ''*eth+*'' ''*eth3+*'' = *Some* ''*eth3+*'' **by** *eval*
lemma *internal-iface-name-wildcard-longest* ''*eth+*'' ''*e+*'' = *Some* ''*eth+*'' **by** *eval*
lemma *internal-iface-name-wildcard-longest* ''*eth+*'' ''*lo*'' = *None* **by** *eval*

lemma *internal-iface-name-wildcard-longest-correct*: *iface-name-is-wildcard i1* $\Longrightarrow$ *iface-name-is-wildcard i2* $\Longrightarrow$
    match-iface (*Iface* (*Pos i1*)) *p-i* $\wedge$ *match-iface* (*Iface* (*Pos i2*)) *p-i* $\longleftrightarrow$
    (*case internal-iface-name-wildcard-longest i1 i2 of None* $\Rightarrow$ *False* | *Some x* $\Rightarrow$ *match-iface* (*Iface* (*Pos x*)) *p-i*)
  apply(*simp split*:*option.split*)
  apply(*intro conjI impI allI*)
   apply(*simp add*: *internal-iface-name-wildcard-longest-def split*: *split-if-asm*)
  apply(*drule match-iface-case-pos-wildcard-prefix*[*of i1 p-i*, *simplified butlast-conv-take match-iface.simps*])
  apply(*drule match-iface-case-pos-wildcard-prefix*[*of i2 p-i*, *simplified butlast-conv-take match-iface.simps*])
   apply (*metis One-nat-def min.commute take-take*)
  apply(*rename-tac x*)
  apply(*simp add*: *internal-iface-name-wildcard-longest-def split*: *split-if-asm*)
   apply(*simp add*: *min-def split*: *split-if-asm*)
   apply(*case-tac internal-iface-name-match x p-i*)
    apply(*simp-all*)
   apply(*frule match-iface-case-pos-wildcard-prefix*[*of i1 p-i*])
   apply(*frule-tac i*=*x* **in** *match-iface-case-pos-wildcard-prefix*[*of - p-i*])
   apply(*simp add*: *butlast-conv-take*)
   apply (*metis min-def take-take*)
  apply(*case-tac internal-iface-name-match x p-i*)
   apply(*simp-all*)
  apply(*frule match-iface-case-pos-wildcard-prefix*[*of i2 p-i*])
  apply(*frule-tac i*=*x* **in** *match-iface-case-pos-wildcard-prefix*[*of - p-i*])

apply(*simp add*: *butlast-conv-take min-def split*:*split-if-asm*)
by (*metis min.commute min-def take-take*)

If the interfaces are no wildcards, they must be equal, otherwise None If one is a wildcard, the other one must 'match', return the non-wildcard If both are wildcards: Longest prefix of both

fun *most-specific-iface* :: *iface* ⇒ *iface* ⇒ *iface option* **where**
most-specific-iface (*Iface* (*Pos i1*)) (*Iface* (*Pos i2*)) = (*case* (*iface-name-is-wildcard i1*, *iface-name-is-wildcard i2*) *of*
(*True*, *True*) ⇒ *map-option* (λ*i*. *Iface* (*Pos i*)) (*internal-iface-name-wildcard-longest i1 i2*) |
(*True*, *False*) ⇒ (*if match-iface* (*Iface* (*Pos i1*)) *i2 then Some* (*Iface* (*Pos i2*)) *else None*) |
(*False*, *True*) ⇒ (*if match-iface* (*Iface* (*Pos i2*)) *i1 then Some* (*Iface* (*Pos i1*)) *else None*) |
(*False*, *False*) ⇒ (*if i1* = *i2 then Some* (*Iface* (*Pos i1*)) *else None*))

definition *all-chars* :: *char list* **where**
all-chars ≡ *Enum.enum*
lemma [*simp*]: *set all-chars* = (*UNIV*::*char set*)
by(*simp add*: *all-chars-def enum-UNIV*)

value (*map* (λ*c*. *c*#''!'') *all-chars*):: *string list*

thm *List.n-lists.simps*

lemma *strings-of-length-n*: *set* (*List.n-lists n all-chars*) = {*s*::*string*. *length s* = *n*}
apply(*induction n*)
apply(*simp*)
apply(*simp*)
apply(*safe*)
apply(*simp*)
apply(*simp*)
apply(*rename-tac n x*)
apply(*rule-tac x*=*drop 1 x* **in** *exI*)
apply(*simp*)
apply(*case-tac x*)
apply(*simp-all*)
done
definition *non-wildcard-ifaces* :: *nat* ⇒ *string list* **where**
non-wildcard-ifaces n ≡ *filter* (λ*i*. ¬ *iface-name-is-wildcard i*) (*List.n-lists n*

all-chars)
export-code *non-wildcard-ifaces* **in** *SML*
lemma *non-wildcard-ifaces*: *set* (*non-wildcard-ifaces n*) = {*s*::*string*. *length s* = *n* ∧ ¬ *iface-name-is-wildcard s*}
using *strings-of-length-n non-wildcard-ifaces-def* **by** *auto*

lemma (⋃ *i* ∈ *set* (*non-wildcard-ifaces n*). *internal-iface-name-to-set i*) = {*s*::*string*. *length s* = *n* ∧ ¬ *iface-name-is-wildcard s*}
apply(*simp-all only*: *internal-iface-name-to-set.simps if-True if-False not-True-eq-False not-False-eq-True non-wildcard-ifaces*)
apply(*simp-all split*: *split-if-asm split-if*)
done

fun *non-wildcard-ifaces-upto* :: *nat* ⇒ *string list* **where**
non-wildcard-ifaces-upto 0 = [[]] |
non-wildcard-ifaces-upto (*Suc n*) = (*non-wildcard-ifaces* (*Suc n*)) @ *non-wildcard-ifaces-upto n*
lemma *non-wildcard-ifaces-upto*: *set* (*non-wildcard-ifaces-upto n*) = {*s*::*string*. *length s* ≤ *n* ∧ ¬ *iface-name-is-wildcard s*}
apply(*induction n*)
apply(*simp*)
apply *fastforce*
apply(*simp add*: *non-wildcard-ifaces*)
by *fastforce*

lemma *inv-i-wildcard*: − {*i*@*cs* | *cs*. *True*} = {*c* | *c*. *length c* < *length i*} ∪ {*c*@*cs* | *c cs*. *length c* = *length i* ∧ *c* ≠ *i*}
apply(*rule*)
prefer *2*
apply(*safe*)[*1*]
apply(*simp add*:)
apply(*simp add*:)
apply(*simp*)
apply(*rule Compl-anti-mono*[**where** *B*={*i* @ *cs* |*cs*. *True*} **and** *A*=− ({*c* | *c*. *length c* < *length i*} ∪ {*c*@*cs* | *c cs*. *length c* = *length i* ∧ *c* ≠ *i*}), *simplified*])
apply(*safe*)
apply(*simp*)
apply(*case-tac* (*length i*) = *length x*)
apply(*erule-tac x*=*x* **in** *allE*, *simp*)
apply(*blast*)
apply(*erule-tac x*=*take* (*length i*) *x* **in** *allE*)
apply(*simp add*: *min-def*)
by (*metis append-take-drop-id*)
lemma *inv-i-nowildcard*: − {*i*::*string*} = {*c* | *c*. *length c* < *length i*} ∪ {*c*@*cs* | *c cs*. *length c* ≥ *length i* ∧ *c* ≠ *i*}
proof −
have *x*: {*c* | *c*. *length c* = *length i* ∧ *c* ≠ *i*} ∪ {*c* | *c*. *length c* > *length i*} = {*c*@*cs* | *c cs*. *length c* ≥ *length i* ∧ *c* ≠ *i*}
apply(*safe*)

apply *force+*
done
have $-$ $\{i::string\} = \{c \mid c\ .\ c \neq i\}$
by(*safe*, *simp*)
also have $\ldots = \{c \mid c.\ length\ c < length\ i\} \cup \{c \mid c.\ length\ c = length\ i \wedge c \neq i\} \cup \{c \mid c.\ length\ c > length\ i\}$
by(*auto*)
finally show *?thesis* **using** *x* **by** *auto*
qed

lemma *inv-iface-name-set*: $-$ (*internal-iface-name-to-set i*) = (
if iface-name-is-wildcard i
then
$\{c \mid c.\ length\ c < length\ (butlast\ i)\} \cup \{c\ @\ cs \mid c\ cs.\ length\ c = length\ (butlast\ i) \wedge c \neq butlast\ i\}$
$(*\{s@cs \mid s\ cs.\ length\ s \leq length\ i - 1 \wedge s \neq take\ (length\ s)\ i\}*)$ (*no ''+'' at end (as one would write donw the iface) but allow arbitrary string*)
else
$\{c \mid c.\ length\ c < length\ i\} \cup \{c@cs \mid c\ cs.\ length\ c \geq length\ i \wedge c \neq i\}$
$(*\ldots \cup X = \ldots \cup \{c \mid c.\ length\ c = length\ i \wedge c \neq i\} \cup \{c \mid c.\ length\ c > length\ i\}$ *and is essentially* $\{c@''+'' \mid len\ c = len\ i \wedge c \neq i\}$
TODO: *this should help when writing as an interface string*)
)
apply(*case-tac iface-name-is-wildcard i*)
apply(*simp-all only*: *internal-iface-name-to-set.simps if-True if-False not-True-eq-False not-False-eq-True*)
apply(*subst inv-i-wildcard*)
apply(*simp*)
apply(*subst inv-i-nowildcard*)
apply(*simp*)
done

lemma $-$ (*internal-iface-name-to-set i*) = (
if iface-name-is-wildcard i
then
$(\bigcup s \in set\ (non\text{-}wildcard\text{-}ifaces\text{-}upto\ (length\ i - 1)).\ internal\text{-}iface\text{-}name\text{-}to\text{-}set\ s) \cup$
$(\bigcup s \in \{c\ @\ cs \mid c\ cs.\ length\ c = length\ (butlast\ i) \wedge c \neq butlast\ i\}.\ internal\text{-}iface\text{-}name\text{-}to\text{-}set\ s)$
else
{} (*TODO*)
)
apply(*subst inv-iface-name-set*)
apply(*case-tac iface-name-is-wildcard i*)
apply(*simp-all only*: *internal-iface-name-to-set.simps if-True if-False not-True-eq-False not-False-eq-True*)

```
  apply(simp)
 oops


 fun neg-iface-name-to-pos-iface-name :: string ⇒ string list where
   neg-iface-name-to-pos-iface-name i = (if iface-name-is-wildcard i
     then
       (non-wildcard-ifaces-upto (length i − 1)) @ map (λs. s@''+'') (filter (λs. s
≠ butlast i) (non-wildcard-ifaces (length i − 1)))
     else
       [] (*TODO*)
   )
 lemma − (internal-iface-name-to-set i) = (⋃ s ∈ set (neg-iface-name-to-pos-iface-name
i). internal-iface-name-to-set s)
   apply(subst inv-iface-name-set)
   apply(subst neg-iface-name-to-pos-iface-name.simps)
   apply(case-tac iface-name-is-wildcard i)
   apply(simp-all only: internal-iface-name-to-set.simps if-True if-False not-True-eq-False
not-False-eq-True)
    apply(simp add: non-wildcard-ifaces-upto non-wildcard-ifaces)
    apply(simp add: iface-name-is-wildcard-alt)
    apply(safe)
    apply(auto)[1]
 oops

hide-const (open) internal-iface-name-wildcard-longest
hide-const (open) internal-iface-name-match

end
theory SimpleFw-Semantics
imports Main ../Common/Negation-Type
 ../Firewall-Common-Decision-State
 ../Primitive-Matchers/IpAddresses
 ../Primitive-Matchers/Iface
 ../Primitive-Matchers/Protocol
 ../Primitive-Matchers/Simple-Packet
begin
```

30 Simple Firewall Syntax (IPv4 only)

```
 datatype simple-action = Accept | Drop
```

Simple match expressions do not allow negated expressions. However, Most match expressions can still be transformed into simple match expressions.

A negated IP address range can be represented as a set of non-negated IP ranges. For example $!8 = \{0..7\} \cup \{8 .. ipv4max\}$. Using CIDR notation (i.e. the $a.b.c.d/n$ notation), we can represent negated IP ranges as a set of non-negated IP ranges with only fair blowup. Another handy result is that the conjunction of two IP ranges in CIDR notation is either the smaller of the two ranges or the empty set. An empty IP range cannot be represented. If one wants to represent the empty range, then the complete rule needs to be removed.

The same holds for layer 4 ports. In addition, there exists an empty port range, e.g. $(1,0)$. The conjunction of two port ranges is again just one port range.

But negation of interfaces is not supported. Since interfaces support a wildcard character, transforming a negated interface would either result in an infeasible blowup or requires knowledge about the existing interfaces (e.g. there only is eth0, eth1, wlan3, and vbox42) An empirical test shows that negated interfaces do not occur in our data sets. Negated interfaces can also be considered bad style: What is !eth0? Everything that is not eth0, experience shows that interfaces may come up randomly, in particular in combination with virtual machines, so !eth0 might not be the desired match. At the moment, if an negated interface occurs which prevents translation to a simple match, we recommend to abstract the negated interface to unknown and remove it (upper or lower closure rule set) before translating to a simple match. The same discussion holds for negated protocols.

Noteworthy, simple match expressions are both expressive and support conjunction: $simple-match1 \wedge simple-match2 = simple-match3$

record *simple-match* =
  iiface :: *iface* — in-interface

  oiface :: *iface* — out-interface
  src :: (*ipv4addr* × *nat*) — source IP address
  dst :: (*ipv4addr* × *nat*) — destination
  proto :: *protocol*
  sports :: (*16 word* × *16 word*) — source-port first:last
  dports :: (*16 word* × *16 word*) — destination-port first:last

datatype *simple-rule* = *SimpleRule* (*match-sel*: *simple-match*) (*action-sel*: *simple-action*)

30.1 Simple Firewall Semantics

fun *simple-match-ip* :: (*ipv4addr* × *nat*) ⇒ *ipv4addr* ⇒ *bool* **where**
  simple-match-ip (*base*, *len*) *p-ip* ⟷ *p-ip* ∈ *ipv4range-set-from-bitmask base len*

— by the way, the words do not wrap around
lemma {(*253*::*8 word*) .. *8*} = {} **by** *simp*

fun *simple-match-port* :: (*16 word* × *16 word*) ⇒ *16 word* ⇒ *bool* **where**
  simple-match-port (*s*,*e*) *p-p* ⟷ *p-p* ∈ {*s*..*e*}

fun *simple-matches* :: *simple-match* ⇒ *simple-packet* ⇒ *bool* **where**

simple-matches m p $\longleftrightarrow$
(*match-iface* (*iiface m*) (*p-iiface p*)) $\wedge$
(*match-iface* (*oiface m*) (*p-oiface p*)) $\wedge$
(*simple-match-ip* (*src m*) (*p-src p*)) $\wedge$
(*simple-match-ip* (*dst m*) (*p-dst p*)) $\wedge$
(*match-proto* (*proto m*) (*p-proto p*)) $\wedge$
(*simple-match-port* (*sports m*) (*p-sport p*)) $\wedge$
(*simple-match-port* (*dports m*) (*p-dport p*))

The semantics of a simple firewall: just iterate over the rules sequentially

fun *simple-fw* :: *simple-rule list* $\Rightarrow$ *simple-packet* $\Rightarrow$ *state* **where**
simple-fw [] - = *Undecided* |
simple-fw ((*SimpleRule m Accept*)#*rs*) *p* = (*if simple-matches m p then Decision FinalAllow else simple-fw rs p*) |
simple-fw ((*SimpleRule m Drop*)#*rs*) *p* = (*if simple-matches m p then Decision FinalDeny else simple-fw rs p*)

definition *simple-match-any* :: *simple-match* **where**
simple-match-any ≡ (| *iiface*=*ifaceAny*, *oiface*=*ifaceAny*, *src*=(*0*,*0*), *dst*=(*0*,*0*), *proto*=*ProtoAny*, *sports*=(*0*,*65535*), *dports*=(*0*,*65535*) |)
lemma *simple-match-any*: *simple-matches simple-match-any p*
proof −
have (*65535*::*16 word*) = *max-word* **by**(*simp add*: *max-word-def*)
thus *?thesis* **by**(*simp add*: *simple-match-any-def ipv4range-set-from-bitmask-0 match-ifaceAny*)
qed

we specify only one empty port range

definition *simple-match-none* :: *simple-match* **where**
simple-match-none ≡ (| *iiface*=*ifaceAny*, *oiface*=*ifaceAny*, *src*=(*1*,*0*), *dst*=(*0*,*0*), *proto*=*ProtoAny*, *sports*=(*1*,*0*), *dports*=(*0*,*65535*) |)
lemma *simple-match-none*: ¬ *simple-matches simple-match-none p*
proof −
show *?thesis* **by**(*simp add*: *simple-match-none-def*)
qed

fun *empty-match* :: *simple-match* $\Rightarrow$ *bool* **where**
empty-match (| *iiface*=-, *oiface*=-, *src*=-, *dst*=-, *proto*=-, *sports*=(*sps1*, *sps2*), *dports*=(*dps1*, *dps2*) |) $\longleftrightarrow$ (*sps1* > *sps2*) ∨ (*dps1* > *dps2*)

lemma *empty-match*: *empty-match m* $\longleftrightarrow$ (∀ *p*. ¬ *simple-matches m p*)
apply(*cases m*, *rename-tac iif oif sip dip protocol sps dps*, *case-tac sps*,*case-tac dps*,*rename-tac sps1 sps2 dps1 dps2*)
apply(*rule iffI*)
apply *fastforce*
apply(*simp*)

proof –
fix *iif oif sip dip protocol sps dps sps1 sps2 dps1 dps2*
let *?x*=λ*p. dps1* $\leq$ *p-dport p* $\longrightarrow$ *p-sport p* $\leq$ *sps2* $\longrightarrow$ *sps1* $\leq$ *p-sport p* $\longrightarrow$

match-proto protocol (*p-proto p*) $\longrightarrow$ *simple-match-ip dip* (*p-dst p*) $\longrightarrow$ *simple-match-ip sip* (*p-src p*) $\longrightarrow$
match-iface oif (*p-oiface p*) $\longrightarrow$ *match-iface iif* (*p-iiface p*) $\longrightarrow$ $\neg$ *p-dport p* $\leq$ *dps2*
assume *m*: *m* = (|*iiface* = *iif*, *oiface* = *oif*, *src* = *sip*, *dst* = *dip*, *proto* = *protocol*, *sports* = (*sps1*, *sps2*), *dports* = (*dps1*, *dps2*)|)
and *nomatch*: $\forall$ *p*::*simple-packet*. *?x p*

have $\bigwedge$*a b. a* $\in$ *ipv4range-set-from-bitmask a b* **using** *ip-set-def ipv4range-set-from-bitmask-eq-ip-set* **by** *blast*
hence *ips*: $\bigwedge$*ips. simple-match-ip ips* (*fst ips*) **by** *force*
have *proto*: *match-proto protocol* (*case protocol of ProtoAny* $\Rightarrow$ *TCP* | *Proto p* $\Rightarrow$ *p*)
by(*simp split*: *protocol.split*)
have *ifaces*: $\bigwedge$*ifce. match-iface ifce* (*iface-sel ifce*)
apply(*case-tac ifce*)
by(*simp add*: *match-iface-refl*)

{ fix *p*::*simple-packet*
from *nomatch* **have** *?x p*
apply –
apply(*erule-tac x*=*p* **in** *allE*)
by *simp*
}note *pkt*=*this*[*of* (|*p-iiface* = *iface-sel iif*,
p-oiface = *iface-sel oif*,
p-src = *fst sip*,
p-dst = *fst dip*,
p-proto = *case protocol of ProtoAny* $\Rightarrow$ *primitive-protocol.TCP* | *Proto p* $\Rightarrow$ *p*,
p-sport = *sps1*,
p-dport = *dps1*|), *simplified*]
from *pkt ips proto ifaces* **have** *sps1* $\leq$ *sps2* $\longrightarrow$ $\neg$ *dps1* $\leq$ *dps2* **by** *blast*

thus *sps2* < *sps1* $\vee$ *dps2* < *dps1* **by** *fastforce*
qed

30.2 Simple Ports

fun *simpl-ports-conjunct* :: (*16 word* $\times$ *16 word*) $\Rightarrow$ (*16 word* $\times$ *16 word*) $\Rightarrow$ (*16 word* $\times$ *16 word*) **where**
simpl-ports-conjunct (*p1s*, *p1e*) (*p2s*, *p2e*) = (*max p1s p2s*, *min p1e p2e*)

lemma {(*p1s*:: *16 word*) .. *p1e*} $\cap$ {*p2s* .. *p2e*} = {*max p1s p2s* .. *min p1e p2e*}
by(*simp*)

lemma *simpl-ports-conjunct-correct*: *simple-match-port p1 pkt* $\wedge$ *simple-match-port p2 pkt* $\longleftrightarrow$ *simple-match-port* (*simpl-ports-conjunct p1 p2*) *pkt*
apply(*cases p1*, *cases p2*, *simp*)
by *blast*

30.3 Simple IPs

lemma *simple-match-ip-conjunct*: *simple-match-ip ip1 p-ip* $\wedge$ *simple-match-ip ip2 p-ip* $\longleftrightarrow$
(*case ipv4cidr-conjunct ip1 ip2 of None* $\Rightarrow$ *False* | *Some ipx* $\Rightarrow$ *simple-match-ip ipx p-ip*)
proof –
{
fix *b1 m1 b2 m2*
have *simple-match-ip* (*b1*, *m1*) *p-ip* $\wedge$ *simple-match-ip* (*b2*, *m2*) *p-ip* $\longleftrightarrow$
p-ip $\in$ *ipv4range-set-from-bitmask b1 m1* $\cap$ *ipv4range-set-from-bitmask b2 m2*
by *simp*
also have ... $\longleftrightarrow$ *p-ip* $\in$ (*case ipv4cidr-conjunct* (*b1*, *m1*) (*b2*, *m2*) *of None* $\Rightarrow$ {} | *Some* (*bx*, *mx*) $\Rightarrow$ *ipv4range-set-from-bitmask bx mx*)
using *ipv4cidr-conjunct-correct* **by** *blast*
also have ... $\longleftrightarrow$ (*case ipv4cidr-conjunct* (*b1*, *m1*) (*b2*, *m2*) *of None* $\Rightarrow$ *False* | *Some ipx* $\Rightarrow$ *simple-match-ip ipx p-ip*)
by(*simp split*: *option.split*)
finally have *simple-match-ip* (*b1*, *m1*) *p-ip* $\wedge$ *simple-match-ip* (*b2*, *m2*) *p-ip* $\longleftrightarrow$
(*case ipv4cidr-conjunct* (*b1*, *m1*) (*b2*, *m2*) *of None* $\Rightarrow$ *False* | *Some ipx* $\Rightarrow$ *simple-match-ip ipx p-ip*) **.**
} **thus** *?thesis* **by**(*cases ip1*, *cases ip2*, *simp*)
qed

declare *simple-matches.simps*[*simp del*]

lemma *nomatch*: $\neg$ *simple-matches m p* $\Longrightarrow$ *simple-fw* (*SimpleRule m a # rs*) *p* = *simple-fw rs p*
by(*cases a*, *simp-all*)

end
theory *SimpleFw-Compliance*
imports *SimpleFw-Semantics ../Primitive-Matchers/Transform*
begin

fun *ipv4-word-netmask-to-ipt-ipv4range* :: (*ipv4addr* $\times$ *nat*) $\Rightarrow$ *ipt-ipv4range* **where**
ipv4-word-netmask-to-ipt-ipv4range (*ip*, *n*) = *Ip4AddrNetmask* (*dotdecimal-of-ipv4addr ip*) *n*

fun *ipt-ipv4range-to-ipv4-word-netmask* :: *ipt-ipv4range* $\Rightarrow$ (*ipv4addr* $\times$ *nat*) **where**

ipt-ipv4range-to-ipv4-word-netmask (*Ip4Addr ip-ddecim*) = (*ipv4addr-of-dotdecimal ip-ddecim*, *32*) |
ipt-ipv4range-to-ipv4-word-netmask (*Ip4AddrNetmask pre len*) = (*ipv4addr-of-dotdecimal pre*, *len*)

30.4 Simple Match to MatchExpr

fun *simple-match-to-ipportiface-match* :: *simple-match* ⇒ *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* (*ipv4-word-netmask-to-ipt-ipv4range sip*)))
(*MatchAnd* (*Match* (*Dst* (*ipv4-word-netmask-to-ipt-ipv4range dip*)))
(*MatchAnd* (*Match* (*Prot p*))
(*MatchAnd* (*Match* (*Src-Ports* [*sps*]))
(*Match* (*Dst-Ports* [*dps*]))
)))))

lemma *matches* γ (*simple-match-to-ipportiface-match* (|*iiface*=*iif*, *oiface*=*oif*, *src*=*sip*, *dst*=*dip*, *proto*=*p*, *sports*=*sps*, *dports*=*dps* |)) *a p* ⟷
matches γ (*alist-and* ([*Pos* (*IIface iif*), *Pos* (*OIface oif*)] @ [*Pos* (*Src* (*ipv4-word-netmask-to-ipt-ipv4range sip*))]
@ [*Pos* (*Dst* (*ipv4-word-netmask-to-ipt-ipv4range dip*))] @ [*Pos* (*Prot p*)]
@ [*Pos* (*Src-Ports* [*sps*])] @ [*Pos* (*Dst-Ports* [*dps*])])) *a p*
apply(*cases sip*,*cases dip*)
apply(*simp add*: *bunch-of-lemmata-about-matches*)
done

lemma *ports-to-set-singleton-simple-match-port*: *p* ∈ *ports-to-set* [*a*] ⟷ *simple-match-port a p*
by(*cases a*, *simp*)

theorem *simple-match-to-ipportiface-match-correct*: *matches* (*common-matcher*, α) (*simple-match-to-ipportiface-match sm*) *a p* ⟷ *simple-matches sm p*
proof −
obtain *iif oif sip dip pro sps dps* **where** *sm*: *sm* = (|*iiface* = *iif*, *oiface* = *oif*, *src* = *sip*, *dst* = *dip*, *proto* = *pro*, *sports* = *sps*, *dports* = *dps*|) **by** (*cases sm*)
{ fix *ip*
have *p-src p* ∈ *ipv4s-to-set* (*ipv4-word-netmask-to-ipt-ipv4range ip*) ⟷ *simple-match-ip ip* (*p-src p*)
and *p-dst p* ∈ *ipv4s-to-set* (*ipv4-word-netmask-to-ipt-ipv4range ip*) ⟷ *simple-match-ip ip* (*p-dst p*)
apply(*case-tac* [!] *ip*)
by(*simp-all add*: *bunch-of-lemmata-about-matches ternary-to-bool-bool-to-ternary ipv4addr-of-dotdecimal-dotdecimal-of-ipv4addr*)

} **note** *simple-match-ips=this*
{ **fix** *ps*
have *p-sport p* $\in$ *ports-to-set* [*ps*] $\longleftrightarrow$ *simple-match-port ps* (*p-sport p*)
and *p-dport p* $\in$ *ports-to-set* [*ps*] $\longleftrightarrow$ *simple-match-port ps* (*p-dport p*)
apply(*case-tac* [!] *ps*)
by(*simp-all*)
} **note** *simple-match-ports=this*
show *?thesis* **unfolding** *sm*
by(*simp add*: *bunch-of-lemmata-about-matches ternary-to-bool-bool-to-ternary simple-match-ips simple-match-ports simple-matches.simps*)
qed

30.5 MatchExpr to Simple Match

30.5.1 Merging Simple Matches

simple-match $\wedge$ *simple-match*

fun *simple-match-and* :: *simple-match* $\Rightarrow$ *simple-match* $\Rightarrow$ *simple-match option*
where
simple-match-and (|*iiface=iif1*, *oiface=oif1*, *src=sip1*, *dst=dip1*, *proto=p1*, *sports=sps1*, *dports=dps1* |)
(|*iiface=iif2*, *oiface=oif2*, *src=sip2*, *dst=dip2*, *proto=p2*, *sports=sps2*, *dports=dps2* |) =
(*case ipv4cidr-conjunct sip1 sip2 of None* $\Rightarrow$ *None* | *Some sip* $\Rightarrow$
(*case ipv4cidr-conjunct dip1 dip2 of None* $\Rightarrow$ *None* | *Some dip* $\Rightarrow$
(*case iface-conjunct iif1 iif2 of None* $\Rightarrow$ *None* | *Some iif* $\Rightarrow$
(*case iface-conjunct oif1 oif2 of None* $\Rightarrow$ *None* | *Some oif* $\Rightarrow$
(*case simple-proto-conjunct p1 p2 of None* $\Rightarrow$ *None* | *Some p* $\Rightarrow$
Some (|*iiface=iif*, *oiface=oif*, *src=sip*, *dst=dip*, *proto=p*,
sports=simpl-ports-conjunct sps1 sps2, *dports=simpl-ports-conjunct dps1 dps2* |))))))

lemma *simple-match-and-correct*: *simple-matches m1 p* $\wedge$ *simple-matches m2 p* $\longleftrightarrow$
(*case simple-match-and m1 m2 of None* $\Rightarrow$ *False* | *Some m* $\Rightarrow$ *simple-matches m p*)
proof −
obtain *iif1 oif1 sip1 dip1 p1 sps1 dps1* **where** *m1*:
m1 = (|*iiface=iif1*, *oiface=oif1*, *src=sip1*, *dst=dip1*, *proto=p1*, *sports=sps1*, *dports=dps1* |) **by**(*cases m1*, *blast*)
obtain *iif2 oif2 sip2 dip2 p2 sps2 dps2* **where** *m2*:
m2 = (|*iiface=iif2*, *oiface=oif2*, *src=sip2*, *dst=dip2*, *proto=p2*, *sports=sps2*, *dports=dps2* |) **by**(*cases m2*, *blast*)

have *sip-None*: *ipv4cidr-conjunct sip1 sip2* = *None* $\Longrightarrow$ $\neg$ *simple-match-ip sip1* (*p-src p*) $\vee$ $\neg$ *simple-match-ip sip2* (*p-src p*)
using *simple-match-ip-conjunct*[*of sip1 p-src p sip2*] **by** *simp*
have *dip-None*: *ipv4cidr-conjunct dip1 dip2* = *None* $\Longrightarrow$ $\neg$ *simple-match-ip dip1* (*p-dst p*) $\vee$ $\neg$ *simple-match-ip dip2* (*p-dst p*)

using *simple-match-ip-conjunct*[*of dip1 p-dst p dip2*] **by** *simp*
have *sip-Some*: $\bigwedge$*ip*. *ipv4cidr-conjunct sip1 sip2* = *Some ip* $\Longrightarrow$
simple-match-ip ip (*p-src p*) $\longleftrightarrow$ *simple-match-ip sip1* (*p-src p*) $\wedge$ *simple-match-ip sip2* (*p-src p*)
using *simple-match-ip-conjunct*[*of sip1 p-src p sip2*] **by** *simp*
have *dip-Some*: $\bigwedge$*ip*. *ipv4cidr-conjunct dip1 dip2* = *Some ip* $\Longrightarrow$
simple-match-ip ip (*p-dst p*) $\longleftrightarrow$ *simple-match-ip dip1* (*p-dst p*) $\wedge$ *simple-match-ip dip2* (*p-dst p*)
using *simple-match-ip-conjunct*[*of dip1 p-dst p dip2*] **by** *simp*

have *iiface-None*: *iface-conjunct iif1 iif2* = *None* $\Longrightarrow$ $\neg$ *match-iface iif1* (*p-iiface p*) $\vee$ $\neg$ *match-iface iif2* (*p-iiface p*)
using *iface-conjunct*[*of iif1* (*p-iiface p*) *iif2*] **by** *simp*
have *oiface-None*: *iface-conjunct oif1 oif2* = *None* $\Longrightarrow$ $\neg$ *match-iface oif1* (*p-oiface p*) $\vee$ $\neg$ *match-iface oif2* (*p-oiface p*)
using *iface-conjunct*[*of oif1* (*p-oiface p*) *oif2*] **by** *simp*
have *iiface-Some*: $\bigwedge$*iface*. *iface-conjunct iif1 iif2* = *Some iface* $\Longrightarrow$
match-iface iface (*p-iiface p*) $\longleftrightarrow$ *match-iface iif1* (*p-iiface p*) $\wedge$ *match-iface iif2* (*p-iiface p*)
using *iface-conjunct*[*of iif1* (*p-iiface p*) *iif2*] **by** *simp*
have *oiface-Some*: $\bigwedge$*iface*. *iface-conjunct oif1 oif2* = *Some iface* $\Longrightarrow$
match-iface iface (*p-oiface p*) $\longleftrightarrow$ *match-iface oif1* (*p-oiface p*) $\wedge$ *match-iface oif2* (*p-oiface p*)
using *iface-conjunct*[*of oif1* (*p-oiface p*) *oif2*] **by** *simp*

have *proto-None*: *simple-proto-conjunct p1 p2* = *None* $\Longrightarrow$ $\neg$ *match-proto p1* (*p-proto p*) $\vee$ $\neg$ *match-proto p2* (*p-proto p*)
using *simple-proto-conjunct-correct*[*of p1* (*p-proto p*) *p2*] **by** *simp*
have *proto-Some*: $\bigwedge$*proto*. *simple-proto-conjunct p1 p2* = *Some proto* $\Longrightarrow$
match-proto proto (*p-proto p*) $\longleftrightarrow$ *match-proto p1* (*p-proto p*) $\wedge$ *match-proto p2* (*p-proto p*)
using *simple-proto-conjunct-correct*[*of p1* (*p-proto p*) *p2*] **by** *simp*

show *?thesis*
apply(*simp add*: *m1 m2*)
apply(*simp split*: *option.split*)
apply(*auto simp add*: *simple-matches.simps*)
apply(*auto dest*: *sip-None dip-None sip-Some dip-Some*)
apply(*auto dest*: *iiface-None oiface-None iiface-Some oiface-Some*)
apply(*auto dest*: *proto-None proto-Some*)
using *simpl-ports-conjunct-correct* **apply**(*blast*)+
done
qed

fun *common-primitive-match-to-simple-match* :: *common-primitive match-expr* $\Rightarrow$ *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 ip*)) = *Some* (*simple-match-any*⦇ *src* := (*ipt-ipv4range-to-ipv4-word-netmask ip*) ⦈) |
common-primitive-match-to-simple-match (*Match* (*Dst ip*)) = *Some* (*simple-match-any*⦇ *dst* := (*ipt-ipv4range-to-ipv4-word-netmask ip*) ⦈) |
common-primitive-match-to-simple-match (*Match* (*Prot p*)) = *Some* (*simple-match-any*⦇ *proto* := *p* ⦈) |
common-primitive-match-to-simple-match (*Match* (*Src-Ports* [])) = *None* |
common-primitive-match-to-simple-match (*Match* (*Src-Ports* [(*s*,*e*)])) = *Some* (*simple-match-any*⦇ *sports* := (*s*,*e*) ⦈) |
common-primitive-match-to-simple-match (*Match* (*Dst-Ports* [])) = *None* |
common-primitive-match-to-simple-match (*Match* (*Dst-Ports* [(*s*,*e*)])) = *Some* (*simple-match-any*⦇ *dports* := (*s*,*e*) ⦈) |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*Prot ProtoAny*))) = *None* |
— TODO:
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*, -) ⇒ *None*
| (-, *None*) ⇒ *None*
| (*Some m1′*, *Some m2′*) ⇒ *simple-match-and m1′ m2′*) |
— undefined cases, normalize before!
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*Prot* -))) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*IIface iif*))) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*OIface oif*))) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*Src* -))) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*Match* (*Dst* -))) = *undefined* |
common-primitive-match-to-simple-match (*MatchNot* (*MatchAnd* - -)) = *undefined* |
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* (*Extra* -)) = *undefined* |

common-primitive-match-to-simple-match (*MatchNot* (*Match* (*Extra* -))) = *undefined*

30.5.2 Normalizing Interfaces

As for now, negated interfaces are simply not allowed

fun *normalized-ifaces* :: *common-primitive match-expr* ⇒ *bool* **where**
normalized-ifaces MatchAny = *True* |
normalized-ifaces (*Match* -) = *True* |
normalized-ifaces (*MatchNot* (*Match* (*IIface* -))) = *False* |
normalized-ifaces (*MatchNot* (*Match* (*OIface* -))) = *False* |
normalized-ifaces (*MatchAnd m1 m2*) = (*normalized-ifaces m1* ∧ *normalized-ifaces m2*) |
normalized-ifaces (*MatchNot* (*MatchAnd* - -)) = *False* |
normalized-ifaces (*MatchNot* -) = *True*

30.5.3 Normalizing Protocols

As for now, negated protocols are simply not allowed

fun *normalized-protocols* :: *common-primitive match-expr* ⇒ *bool* **where**
normalized-protocols MatchAny = *True* |
normalized-protocols (*Match* -) = *True* |
normalized-protocols (*MatchNot* (*Match* (*Prot* -))) = *False* |
normalized-protocols (*MatchAnd m1 m2*) = (*normalized-protocols m1* ∧ *normalized-protocols m2*) |
normalized-protocols (*MatchNot* (*MatchAnd* - -)) = *False* |
normalized-protocols (*MatchNot* -) = *True*

lemma *match-iface-simple-match-any-simps*:
match-iface (*iiface simple-match-any*) (*p-iiface 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*)
apply(*simp-all add*: *simple-match-any-def match-ifaceAny ipv4range-set-from-bitmask-0*)
apply(*subgoal-tac* [!] (*65535*::*16 word*) = *max-word*)
apply(*simp-all*)
apply(*simp-all add*: *max-word-def*)
done

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-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 –
{ **fix** *ip*
have *p-src p* $\in$ *ipv4s-to-set ip* $\longleftrightarrow$ *simple-match-ip* (*ipt-ipv4range-to-ipv4-word-netmask ip*) (*p-src p*)
and *p-dst p* $\in$ *ipv4s-to-set ip* $\longleftrightarrow$ *simple-match-ip* (*ipt-ipv4range-to-ipv4-word-netmask ip*) (*p-dst p*)
by(*case-tac* [!] *ip*)(*simp-all add*: *ipv4range-set-from-bitmask-32*)
}**note** *matches-SrcDst-simple-match2*=*this*
show *?thesis*
using *assms* **proof**(*induction m arbitrary*: *sm rule*: *common-primitive-match-to-simple-match.induct*)
case *1* **thus** *?case*
by(*simp-all add*: *match-iface-simple-match-any-simps bunch-of-lemmata-about-matches*(*2*) *simple-matches.simps*)
next
case (*13 m1 m2*)
let *?caseSome*=*Some sm* = *common-primitive-match-to-simple-match* (*MatchAnd m1 m2*)
let *?caseNone*=*common-primitive-match-to-simple-match* (*MatchAnd m1 m2*) = *None*
let *?goal*=(*?caseSome* $\longrightarrow$ *matches* (*common-matcher*, α) (*MatchAnd m1 m2*) *a p* = *simple-matches sm p*) $\wedge$
(*?caseNone* $\longrightarrow$ $\neg$ *matches* (*common-matcher*, α) (*MatchAnd m1 m2*) *a p*)

{ **assume** *caseNone*: *?caseNone*
{ **fix** *sm1 sm2*
assume *sm1*: *common-primitive-match-to-simple-match m1* = *Some sm1*
and *sm2*: *common-primitive-match-to-simple-match m2* = *Some sm2*
and *sma*: *simple-match-and sm1 sm2* = *None*
from *sma simple-match-and-correct* **have** *1*: $\neg$ (*simple-matches sm1 p* $\wedge$ *simple-matches sm2 p*) **by** *simp*
from *sm1 sm2 13* **have** *2*: (*matches* (*common-matcher*, α) *m1 a p* $\longleftrightarrow$ *simple-matches sm1 p*) $\wedge$
(*matches* (*common-matcher*, α) *m2 a p* $\longleftrightarrow$ *simple-matches sm2 p*) **by** *force*
hence *2*: *matches* (*common-matcher*, α) (*MatchAnd m1 m2*) *a p* $\longleftrightarrow$ *simple-matches sm1 p* $\wedge$ *simple-matches sm2 p*
by(*simp add*: *bunch-of-lemmata-about-matches*)
from *1 2* **have** $\neg$ *matches* (*common-matcher*, α) (*MatchAnd m1 m2*) *a p* **by** *blast*
}

with *caseNone* **have** *common-primitive-match-to-simple-match m1* = *None* ∨
common-primitive-match-to-simple-match m2 = *None* ∨
¬ *matches* (*common-matcher*, α) (*MatchAnd m1 m2*) *a p*
by(*simp split*:*option.split-asm*)
hence ¬ *matches* (*common-matcher*, α) (*MatchAnd m1 m2*) *a p*
apply(*elim disjE*)
apply(*simp-all*)
using *13* **apply**(*simp-all add*: *bunch-of-lemmata-about-matches*(*1*))
done
}**note** *caseNone*=*this*

{ **assume** *caseSome*: *?caseSome*
hence ∃ *sm1*. *common-primitive-match-to-simple-match m1* = *Some sm1* **and**
∃ *sm2*. *common-primitive-match-to-simple-match m2* = *Some sm2*
by(*simp-all split*: *option.split-asm*)
from *this* **obtain** *sm1 sm2* **where** *sm1*: *Some sm1* = *common-primitive-match-to-simple-match m1*
and *sm2*: *Some sm2* = *common-primitive-match-to-simple-match m2* **by** *fastforce*+
with *13* **have** *matches* (*common-matcher*, α) *m1 a p* = *simple-matches sm1 p* ∧
matches (*common-matcher*, α) *m2 a p* = *simple-matches sm2 p* **by** *simp*
hence *1*: *matches* (*common-matcher*, α) (*MatchAnd m1 m2*) *a p* ⟷ *simple-matches sm1 p* ∧ *simple-matches sm2 p*
by(*simp add*: *bunch-of-lemmata-about-matches*)
from *caseSome sm1 sm2* **have** *simple-match-and sm1 sm2* = *Some sm* **by**(*simp split*: *option.split-asm*)
with *simple-match-and-correct* **have** *2*: *simple-matches sm p* ⟷ *simple-matches sm1 p* ∧ *simple-matches sm2 p* **by** *simp*
from *1 2* **have** *matches* (*common-matcher*, α) (*MatchAnd m1 m2*) *a p* = *simple-matches sm p* **by** *simp*
} **note** *caseSome*=*this*

from *caseNone caseSome* **show** *?goal* **by** *blast*
qed(*simp-all add*: *match-iface-simple-match-any-simps simple-matches.simps*,
simp-all add: *bunch-of-lemmata-about-matches ternary-to-bool-bool-to-ternary matches-SrcDst-simple-match2*)
qed

fun *action-to-simple-action* :: *action* ⇒ *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* :: *common-primitive rule list* ⇒ *bool* **where**
check-simple-fw-preconditions rs ≡ ∀ *r* ∈ *set rs*. (*case r of* (*Rule m a*) ⇒ *normalized-src-ports m* ∧ *normalized-dst-ports m* ∧ *normalized-src-ips m* ∧ *normalized-dst-ips m* ∧ *normalized-ifaces m* ∧
normalized-protocols m ∧ ¬ *has-disc is-Extra m* ∧ (*a* = *action.Accept* ∨ *a* = *action.Drop*))
definition *to-simple-firewall* :: *common-primitive rule list* ⇒ *simple-rule list* **where**
to-simple-firewall rs ≡ *List.map-filter* (λ*r*. *case r of Rule m a* ⇒
(*case* (*common-primitive-match-to-simple-match m*) *of None* ⇒ *None* |
Some sm ⇒ *Some* (*SimpleRule sm* (*action-to-simple-action a*)))) *rs*

lemma *to-simple-firewall-simps*:
to-simple-firewall [] = []
to-simple-firewall ((*Rule m a*)#*rs*) = (*case common-primitive-match-to-simple-match m of*
None ⇒ *to-simple-firewall rs*
| *Some sm* ⇒ (*SimpleRule sm* (*action-to-simple-action a*)) # *to-simple-firewall rs*)
by(*simp-all add*: *to-simple-firewall-def List.map-filter-simps split*: *option.split*)

value *check-simple-fw-preconditions*
[*Rule* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*127*, *0*, *0*, *0*) *8*)))
(*MatchAnd* (*Match* (*Dst-Ports* [(*0*, *65535*)]))
(*Match* (*Src-Ports* [(*0*, *65535*)]))))
Drop]
value *to-simple-firewall*
[*Rule* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*127*, *0*, *0*, *0*) *8*)))
(*MatchAnd* (*Match* (*Dst-Ports* [(*0*, *65535*)]))
(*Match* (*Src-Ports* [(*0*, *65535*)]))))
Drop]
value *check-simple-fw-preconditions* [*Rule* (*MatchAnd MatchAny MatchAny*) *Drop*]
value *to-simple-firewall* [*Rule* (*MatchAnd MatchAny MatchAny*) *Drop*]
value *to-simple-firewall* [*Rule* (*Match* (*Src* (*Ip4AddrNetmask* (*127*, *0*, *0*, *0*) *8*))) *Drop*]

theorem *to-simple-firewall*: *check-simple-fw-preconditions rs* ⟹ *approximating-bigstep-fun* (*common-matcher*, α) *p rs Undecided* = *simple-fw* (*to-simple-firewall rs*) *p*
proof(*induction rs*)
case *Nil* **thus** *?case* **by**(*simp add*: *to-simple-firewall-simps*)
next
case (*Cons r rs*)
from *Cons* **have** *IH*: *approximating-bigstep-fun* (*common-matcher*, α) *p rs Undecided* = *simple-fw* (*to-simple-firewall rs*) *p*
by(*simp add*: *check-simple-fw-preconditions-def*)
obtain *m a* **where** *r*: *r* = *Rule m a* **by**(*cases r*, *simp*)
from *Cons.prems* **have** *check-simple-fw-preconditions* [*r*] **by**(*simp add*: *check-simple-fw-preconditions-def*)
with *r common-primitive-match-to-simple-match*

have *match*: $\bigwedge$ *sm*. *common-primitive-match-to-simple-match m* = *Some sm* $\Longrightarrow$ *matches* (*common-matcher*, α) *m a p* = *simple-matches sm p* **and**
nomatch: *common-primitive-match-to-simple-match m* = *None* $\Longrightarrow$ ¬ *matches* (*common-matcher*, α) *m a p*
unfolding *check-simple-fw-preconditions-def* **by** *simp-all*
from ‹*check-simple-fw-preconditions* [*r*]› **have** *a* = *action.Accept* ∨ *a* = *action.Drop* **by**(*simp add*: *r check-simple-fw-preconditions-def*)
thus *?case*
by(*auto simp add*: *r to-simple-firewall-simps IH match nomatch split*: *option.split action.split*)
qed

end
theory *Shadowed*
imports *SimpleFw-Semantics*
../Common/Negation-Type-DNF
../Primitive-Matchers/Ports
begin

31 Optimizing Simple Firewall

31.1 Removing Shadowed Rules

Assumes: *simple-ruleset*

fun *rmshadow* :: *simple-rule list* ⇒ *simple-packet set* ⇒ *simple-rule list* **where**
rmshadow [] - = [] |
rmshadow ((*SimpleRule m a*)#*rs*) *P* = (*if* (∀ *p*∈*P*. ¬ *simple-matches m p*)
then
rmshadow rs P
else
(*SimpleRule m a*) # (*rmshadow rs* {*p* ∈ *P*. ¬ *simple-matches m p*}))

31.1.1 Soundness

lemma *rmshadow-sound*:
p ∈ *P* $\Longrightarrow$ *simple-fw* (*rmshadow rs P*) *p* = *simple-fw rs p*
proof(*induction rs arbitrary*: *P*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
from *Cons.IH Cons.prems* **have** *IH1*: *simple-fw* (*rmshadow rs P*) *p* = *simple-fw rs p* **by** (*simp*)
let *?P'*={*p* ∈ *P*. ¬ *simple-matches* (*match-sel r*) *p*}
from *Cons.IH Cons.prems* **have** *IH2*: $\bigwedge$*m*. *p* ∈ *?P'* $\Longrightarrow$ *simple-fw* (*rmshadow rs ?P'*) *p* = *simple-fw rs p* **by** *simp*
from *Cons.prems* **show** *?case*
apply(*cases r*, *rename-tac m a*)
apply(*simp*)
apply(*case-tac* ∀ *p*∈*P*. ¬ *simple-matches m p*)

apply(*simp add*: *IH1 nomatch*)
apply(*case-tac* $p \in$ *?P'*)
apply(*frule IH2*)
apply(*simp add*: *nomatch IH1*)
apply(*simp*)
apply(*case-tac a*)
apply(*simp-all*)
by *fast+*
qed

A different approach where we start with the empty set of packets and collect packets which are already "matched-away".

fun *rmshadow'* :: *simple-rule list* ⇒ *simple-packet set* ⇒ *simple-rule list* **where**
rmshadow' [] - = [] |
rmshadow' ((*SimpleRule m a*)#*rs*) *P* = (*if* {*p*. *simple-matches m p*} ⊆ *P*
then
rmshadow' rs P
else
(*SimpleRule m a*) # (*rmshadow' rs* (*P* ∪ {*p*. *simple-matches m p*})))

lemma *rmshadow'-sound*:
p ∉ *P* ⟹ *simple-fw* (*rmshadow' rs P*) *p* = *simple-fw rs p*
proof(*induction rs arbitrary*: *P*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons r rs*)
from *Cons.IH Cons.prems* **have** *IH1*: *simple-fw* (*rmshadow' rs P*) *p* = *simple-fw rs p* **by** (*simp*)
let *?P'*={*p*. *simple-matches* (*match-sel r*) *p*}
from *Cons.IH Cons.prems* **have** *IH2*: ⋀*m*. *p* ∉ (*Collect* (*simple-matches m*)) ⟹ *simple-fw* (*rmshadow' rs* (*P* ∪ *Collect* (*simple-matches m*))) *p* = *simple-fw rs p* **by** *simp*
have *nomatch-m*: ⋀*m*. *p* ∉ *P* ⟹ {*p*. *simple-matches m p*} ⊆ *P* ⟹ ¬ *simple-matches m p* **by** *blast*
from *Cons.prems* **show** *?case*
apply(*cases r*, *rename-tac m a*)
apply(*simp*)
apply(*case-tac* {*p*. *simple-matches m p*} ⊆ *P*)
apply(*simp add*: *IH1*)
apply(*drule-tac m*=*m* **in** *nomatch-m*)
apply(*simp*)
apply(*simp add*: *nomatch*)
apply(*simp*)
apply(*case-tac a*)
apply(*simp-all*)
apply(*simp-all add*: *IH2*)
done
qed

corollary *simple-fw* (*rmshadow rs UNIV*) *p* = *simple-fw* (*rmshadow′ rs* {}) *p*
using *rmshadow′-sound rmshadow-sound* **by** *auto*

value *rmshadow* [*SimpleRule* (|*iiface* = *Iface* ′′+′′, *oiface* = *Iface* ′′+′′, *src* = (*0*, *0*), *dst* = (*0*, *0*), *proto* = *Proto TCP*, *sports* = (*0*, *0xFFFF*), *dports* = (*0x16*, *0x16*)|)
simple-action.Drop,
SimpleRule (|*iiface* = *Iface* ′′+′′, *oiface* = *Iface* ′′+′′, *src* = (*0*, *0*), *dst* = (*0*, *0*), *proto* = *ProtoAny*, *sports* = (*0*, *0xFFFF*), *dports* = (*0*, *0xFFFF*)|)
simple-action.Accept,
SimpleRule (|*iiface* = *Iface* ′′+′′, *oiface* = *Iface* ′′+′′, *src* = (*0*, *0*), *dst* = (*0*, *0*), *proto* = *Proto TCP*, *sports* = (*0*, *0xFFFF*), *dports* = (*0x138E*, *0x138E*)|)
simple-action.Drop] *UNIV*

Previous algorithm is not executable because we have no code for *simple-packet set*. To get some code, some efficient set operations would be necessary. We either need union and subset or intersection and negation. Both subset and negation are complicated. Probably the BDDs which related work uses is really necessary.

context
begin
private type-synonym *simple-packet-set* = *simple-match list*

private definition *simple-packet-set-toSet* :: *simple-packet-set* ⇒ *simple-packet set* **where**
simple-packet-set-toSet ms = {*p*. ∃ *m* ∈ *set ms*. *simple-matches m p*}

private lemma *simple-packet-set-toSet-alt*: *simple-packet-set-toSet ms* = (⋃ *m* ∈ *set ms*. {*p*. *simple-matches m p*})
unfolding *simple-packet-set-toSet-def* **by** *blast*

private definition *simple-packet-set-union* :: *simple-packet-set* ⇒ *simple-match* ⇒ *simple-packet-set* **where**
simple-packet-set-union ps m = *m* # *ps*

private lemma *simple-packet-set-toSet* (*simple-packet-set-union ps m*) = *simple-packet-set-toSet ps* ∪ {*p*. *simple-matches m p*}
unfolding *simple-packet-set-toSet-def simple-packet-set-union-def* **by** *simp blast*

value (∃ *m′* ∈ *set ms*. *iface-subset iif* (*iiface m′*)) ∧
(∃ *m′* ∈ *set ms*. *iface-subset oif* (*oiface m′*)) ∧
ipv4range-subset (*ipv4-cidr-tuple-to-interval sip*) (*l2br* (*map ipv4cidr-to-interval* (*map src ms*)))

private lemma (∃ *m′* ∈ *set ms*.
{*i*. *match-iface iif i*} ⊆ {*i*. *match-iface* (*iiface m′*) *i*} ∧
{*i*. *match-iface oif i*} ⊆ {*i*. *match-iface* (*oiface m′*) *i*} ∧

$\{ip.\ simple\text{-}match\text{-}ip\ sip\ ip\} \subseteq \{ip.\ simple\text{-}match\text{-}ip\ (src\ m')\ ip\} \wedge$
$\{ip.\ simple\text{-}match\text{-}ip\ dip\ ip\} \subseteq \{ip.\ simple\text{-}match\text{-}ip\ (dst\ m')\ ip\} \wedge$
$\{p.\ match\text{-}proto\ protocol\ p\} \subseteq \{p.\ match\text{-}proto\ (proto\ m')\ p\} \wedge$
$\{p.\ simple\text{-}match\text{-}port\ sps\ p\} \subseteq \{p.\ simple\text{-}match\text{-}port\ (sports\ m')\ p\} \wedge$
$\{p.\ simple\text{-}match\text{-}port\ dps\ p\} \subseteq \{p.\ simple\text{-}match\text{-}port\ (dports\ m')\ p\}$
)
$\Longrightarrow \{p.\ simple\text{-}matches\ (\!|iiface{=}iif,\ oiface{=}oif,\ src{=}sip,\ dst{=}dip,\ proto{=}protocol,\ sports{=}sps,\ dports{=}dps\ |\!)\ p\} \subseteq (simple\text{-}packet\text{-}set\text{-}toSet\ ms)$
unfolding *simple-packet-set-toSet-def simple-packet-set-union-def*
apply(*simp add*: *simple-matches.simps*)
apply(*simp add*: *Set.Collect-mono-iff*)
apply *clarify*
apply *meson*
done

subset or negation ... One efficient implementation would suffice.

private lemma $\{p.\ simple\text{-}matches\ m\ p\} \subseteq (simple\text{-}packet\text{-}set\text{-}toSet\ ms) \longleftrightarrow \{p.\ simple\text{-}matches\ m\ p\} \cap (\bigcap\ m \in set\ ms.\ \{p.\ \neg\ simple\text{-}matches\ m\ p\}) = \{\}$ (**is** $?l \longleftrightarrow ?r$)
proof −
have $?l \longleftrightarrow \{p.\ simple\text{-}matches\ m\ p\} - (simple\text{-}packet\text{-}set\text{-}toSet\ ms) = \{\}$ **by** *blast*
also have $\ldots \longleftrightarrow \{p.\ simple\text{-}matches\ m\ p\} - (\bigcup\ m \in set\ ms.\ \{p.\ simple\text{-}matches\ m\ p\}) = \{\}$
using *simple-packet-set-toSet-alt* **by** *simp*
also have $\ldots \longleftrightarrow ?r$ **by** *blast*
finally show *?thesis* **.**
qed

end
end