Iptables-Semantics

Cornelius Diekmann, Lars Hupel

January 5, 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-new *'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* $xs_1 \leq$ *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* $\longleftrightarrow$ γ *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*
$(\text{-},\text{-},\text{-}\vdash \langle \text{-}, \text{-}\rangle \Rightarrow \text{-}\ \ [60,60,60,20,98,98]\ 89)$
for Γ **and** γ **and** *p* **where**
skip: $\Gamma,\gamma,p\vdash \langle [],\ t\rangle \Rightarrow t$ |
accept: *matches* γ *m p* $\Longrightarrow$ $\Gamma,\gamma,p\vdash \langle$[*Rule m Accept*], *Undecided*$\rangle \Rightarrow$ *Decision FinalAllow* |
drop: *matches* γ *m p* $\Longrightarrow$ $\Gamma,\gamma,p\vdash \langle$[*Rule m Drop*], *Undecided*$\rangle \Rightarrow$ *Decision FinalDeny* |
reject: *matches* γ *m p* $\Longrightarrow$ $\Gamma,\gamma,p\vdash \langle$[*Rule m Reject*], *Undecided*$\rangle \Rightarrow$ *Decision FinalDeny* |
log: *matches* γ *m p* $\Longrightarrow$ $\Gamma,\gamma,p\vdash \langle$[*Rule m Log*], *Undecided*$\rangle \Rightarrow$ *Undecided* |

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

$\Gamma,\gamma,p \vdash \langle [Rule\ m\ (Call\ chain)],\ Undecided \rangle \Rightarrow Undecided$ |
call-result: $[\![$ $matches\ \gamma\ m\ p$; $\Gamma\ chain = Some\ rs$; $\Gamma,\gamma,p \vdash \langle rs,\ Undecided \rangle \Rightarrow t$ $]\!]$ $\Longrightarrow$
$\Gamma,\gamma,p \vdash \langle [Rule\ m\ (Call\ chain)],\ Undecided \rangle \Rightarrow t$

The semantic rules again in pretty format:

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

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

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

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

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

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

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

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

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

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

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

lemma *deny*:
$matches\ \gamma\ m\ p \Longrightarrow a = Drop \lor a = Reject \Longrightarrow iptables\text{-}bigstep\ \Gamma\ \gamma\ p\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalDeny)$
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*]:
$[\![$ $\Gamma,\gamma,p\vdash \langle rs,s\rangle \Rightarrow t$;
$\bigwedge t.\ P\ []\ t\ t$;
$\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Accept \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalAllow)$;
$\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Drop \vee a = Reject \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalDeny)$;
$\bigwedge m\ a.\ matches\ \gamma\ m\ p \Longrightarrow a = Log \vee a = Empty \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided$;
$\bigwedge m\ a.\ \neg\ matches\ \gamma\ m\ p \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided$;
$\bigwedge rs\ X.\ P\ rs\ (Decision\ X)\ (Decision\ X)$;
$\bigwedge rs\ rs_1\ rs_2\ t\ t'.\ rs = rs_1\ @\ rs_2 \Longrightarrow \Gamma,\gamma,p\vdash \langle rs_1, Undecided\rangle \Rightarrow t \Longrightarrow P\ rs_1\ Undecided\ t \Longrightarrow \Gamma,\gamma,p\vdash \langle rs_2,t\rangle \Rightarrow t' \Longrightarrow P\ rs_2\ t\ t' \Longrightarrow P\ rs\ Undecided\ t'$;
$\bigwedge m\ a\ chain\ rs_1\ m'\ rs_2.\ matches\ \gamma\ m\ p \Longrightarrow a = Call\ chain \Longrightarrow \Gamma\ chain = Some\ (rs_1\ @\ [Rule\ m'\ Return]\ @\ rs_2) \Longrightarrow matches\ \gamma\ m'\ p \Longrightarrow \Gamma,\gamma,p\vdash \langle rs_1, Undecided\rangle \Rightarrow Undecided \Longrightarrow P\ rs_1\ Undecided\ Undecided \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided$;
$\bigwedge m\ a\ chain\ rs\ t.\ matches\ \gamma\ m\ p \Longrightarrow a = Call\ chain \Longrightarrow \Gamma\ chain = Some\ rs \Longrightarrow \Gamma,\gamma,p\vdash \langle rs, Undecided\rangle \Rightarrow t \Longrightarrow P\ rs\ Undecided\ t \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ t$ $]\!] \Longrightarrow$
$P\ rs\ s\ t$
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*: $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t \Longrightarrow r = [\mathit{Rule}\ m\ \mathit{Log}] \Longrightarrow \mathit{matches}\ \gamma\ m\ p \Longrightarrow s = \mathit{Undecided} \Longrightarrow t = \mathit{Undecided}$
by (*induction rule*: *iptables-bigstep.induct*) *auto*

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

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

lemma *callD*:
  assumes $\Gamma,\gamma,p\vdash \langle r,\ s\rangle \Rightarrow t$ $r = [\mathit{Rule}\ m\ (\mathit{Call}\ \mathit{chain})]$ $s = \mathit{Undecided}$ $\mathit{matches}\ \gamma\ m\ p$ $\Gamma\ \mathit{chain} = \mathit{Some}\ \mathit{rs}$
  obtains $\Gamma,\gamma,p\vdash \langle \mathit{rs},s\rangle \Rightarrow t$
    | $\mathit{rs}_1\ \mathit{rs}_2\ m'$ **where** $\mathit{rs} = \mathit{rs}_1$ @ $\mathit{Rule}\ m'\ \mathit{Return}$ # rs_2 $\mathit{matches}\ \gamma\ m'\ p$ $\Gamma,\gamma,p\vdash \langle \mathit{rs}_1,s\rangle \Rightarrow \mathit{Undecided}$ $t = \mathit{Undecided}$
  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 $\mathit{rs} = \mathit{rs}_1$ @ rs_2 $\Gamma,\gamma,p\vdash \langle \mathit{rs}_1,s\rangle \Rightarrow t$ $\Gamma,\gamma,p\vdash \langle \mathit{rs}_2,t\rangle \Rightarrow t'$
  shows $\Gamma,\gamma,p\vdash \langle \mathit{rs},s\rangle \Rightarrow t'$
using *assms* **by** (*cases s*) (*auto intro*: *seq decision dest*: *decisionD*)

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

lemma *seq-split*:
  assumes $\Gamma,\gamma,p\vdash \langle \mathit{rs},\ s\rangle \Rightarrow t$ $\mathit{rs} = \mathit{rs}_1@\mathit{rs}_2$
  obtains t' **where** $\Gamma,\gamma,p\vdash \langle \mathit{rs}_1,s\rangle \Rightarrow t'$ $\Gamma,\gamma,p\vdash \langle \mathit{rs}_2,t'\rangle \Rightarrow t$
  using *assms*
  proof (*induction rs s t arbitrary*: $\mathit{rs}_1\ \mathit{rs}_2$ *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 rs1) (auto intro: iptables-bigstep.intros)
  next
    case (Seq rs rsa rsb t t')
    hence rs: rsa @ rsb = rs1 @ rs2 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: Γ,γ,p⊢ ⟨take (length rsa) rs1, Undecided⟩ ⇒ t
          by simp
        from Seq longer obtain t2
          where t2a: Γ,γ,p⊢ ⟨drop (length rsa) rs1,t⟩ ⇒ t2
            and rs2-t2: Γ,γ,p⊢ ⟨rs2,t2⟩ ⇒ t'
          by blast
          with t1 rs2-t2 have Γ,γ,p⊢ ⟨take (length rsa) rs1 @ drop (length rsa)
rs1,Undecided⟩ ⇒ t2
          by (blast intro: iptables-bigstep.seq)
        with Seq rs2-t2 show ?thesis
          by simp
      next
        case shorter
        with rs have rsa': rsa = rs1 @ take (length rsa − length rs1) rs2
          by (metis append-eq-conv-conj length-drop)
        from shorter rs have rsb': rsb = drop (length rsa − length rs1) rs2
          by (metis append-eq-conv-conj length-drop)
        from Seq rsa' obtain t1
          where t1a: Γ,γ,p⊢ ⟨rs1,Undecided⟩ ⇒ t1
            and t1b: Γ,γ,p⊢ ⟨take (length rsa − length rs1) rs2,t1⟩ ⇒ t
          by blast
         from rsb' Seq.hyps have t2: Γ,γ,p⊢ ⟨drop (length rsa − length rs1) rs2,t⟩
⇒ t'
          by blast
        with seq' t1b have Γ,γ,p⊢ ⟨rs2,t1⟩ ⇒ t'
          by fastforce
        with Seq t1a show ?thesis
          by fast
      qed
  next
    case Call-return
     hence Γ,γ,p⊢ ⟨rs1, Undecided⟩ ⇒ Undecided Γ,γ,p⊢ ⟨rs2, Undecided⟩ ⇒
Undecided
   by (case-tac [!] rs1) (auto intro: iptables-bigstep.skip iptables-bigstep.call-return)
    thus ?case by fact
  next
    case (Call-result - - - - t)
    show ?case
      proof (cases rs1)
        case Nil
         with Call-result have Γ,γ,p⊢ ⟨rs1, Undecided⟩ ⇒ Undecided Γ,γ,p⊢ ⟨rs2,
```

Undecided⟩ ⇒ *t*
        by (*auto intro*: *iptables-bigstep.intros*)
      thus *?thesis* **by** *fact*
    next
      case *Cons*
      with *Call-result* **have** $\Gamma,\gamma,p\vdash \langle rs_1,\ Undecided\rangle \Rightarrow t$ $\Gamma,\gamma,p\vdash \langle rs_2,\ t\rangle \Rightarrow t$
        by (*auto intro*: *iptables-bigstep.intros*)
      thus *?thesis* **by** *fact*
    qed
  qed (*auto intro*: *iptables-bigstep.intros*)

lemma *seqE*:
  assumes $\Gamma,\gamma,p\vdash \langle rs_1 @ rs_2,\ s\rangle \Rightarrow t$
  obtains *ti* **where** $\Gamma,\gamma,p\vdash \langle rs_1,s\rangle \Rightarrow ti$ $\Gamma,\gamma,p\vdash \langle rs_2,ti\rangle \Rightarrow t$
  using *assms* **by** (*force elim*: *seq-split*)

lemma *seqE-cons*:
  assumes $\Gamma,\gamma,p\vdash \langle r\#rs,\ s\rangle \Rightarrow t$
  obtains *ti* **where** $\Gamma,\gamma,p\vdash \langle [r],s\rangle \Rightarrow ti$ $\Gamma,\gamma,p\vdash \langle rs,ti\rangle \Rightarrow t$
  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*)

lemma *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*

lemma *no-free-return*: $\Gamma,\gamma,p\vdash \langle[Rule\ m\ Return],\ Undecided\rangle \Rightarrow t \Longrightarrow matches\ \gamma\ m\ p \Longrightarrow False$
  by (*metis no-free-return-hlp*)

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*: $\Gamma,\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow t' \Longrightarrow matches\ \gamma\ m\ p$

$\Longrightarrow$

Γ *chain* = *Some* (*rsa* @ *Rule* m' *Return* # *rsb*) $\Longrightarrow$
matches γ 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*)

lemma *no-free-return-seq*:
assumes $\Gamma,\gamma,p\vdash$ $\langle$*r1* @ *Rule m Return* # *r2*, *Undecided*$\rangle$ $\Rightarrow$ *t matches* γ *m p* $\Gamma,\gamma,p\vdash$ $\langle$*r1*,*Undecided*$\rangle$ $\Rightarrow$ *Undecided*

shows *False*
proof –
from *assms* **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*
with *assms* **show** *False*
by (*blast intro*: *no-free-return elim*: *seq-split*)
qed

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-fst*:
assumes $\Gamma,\gamma,p\vdash \langle r\#rs,\ s\rangle \Rightarrow t$
obtains (*decision*) X **where** $s = Decision\ X$
| (*nomatch*) $m\ a$ **where** $r = Rule\ m\ a\ a \neq Return \vee \neg\ matches\ \gamma\ m\ p$
using *assms*
proof (*induction r#rs s t rule*: *iptables-bigstep-induct*)
case *Seq* **thus** *?case*
by (*metis no-free-return-seq seq skip rule.exhaust*)
qed *auto*

lemma *iptables-bigstep-deterministic*: $[\![\ \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t;\ \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t'\ ]\!]$
$\Longrightarrow t = t'$
proof (*induction arbitrary*: t' *rule*: *iptables-bigstep-induct*)
case *Seq*
thus *?case*
by (*metis seq-split*)
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*)

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

lemma *iptables-bigstep-to-decision*: $\Gamma,\gamma,p\vdash \langle rs,\ Decision\ Y\rangle \Rightarrow Decision\ X \Longrightarrow Y$
$= X$
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 $\Gamma,\gamma,p\vdash \langle rs_1\ @\ [\mathit{Rule}\ m\ \mathit{Log}]\ @\ rs_2,\ s\rangle \Rightarrow t$
shows $\Gamma,\gamma,p\vdash \langle rs_1\ @\ rs_2,\ s\rangle \Rightarrow t$
proof −
from *assms* **obtain** t' **where** t': $\Gamma,\gamma,p\vdash \langle rs_1,\ s\rangle \Rightarrow t'$ $\Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Log}]\ @\ rs_2,\ t'\rangle \Rightarrow t$
by (*blast elim*: *seqE*)
hence $\Gamma,\gamma,p\vdash \langle \mathit{Rule}\ m\ \mathit{Log}\ \#\ rs_2,\ t'\rangle \Rightarrow t$
by *simp*
then obtain t'' **where** $\Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Log}],\ t'\rangle \Rightarrow t''$ $\Gamma,\gamma,p\vdash \langle rs_2,\ t''\rangle \Rightarrow t$
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 $\Gamma,\gamma,p\vdash \langle rs_1\ @\ [\mathit{Rule}\ m\ \mathit{Empty}]\ @\ rs_2,\ s\rangle \Rightarrow t$
shows $\Gamma,\gamma,p\vdash \langle rs_1\ @\ rs_2,\ s\rangle \Rightarrow t$
proof −
from *assms* **obtain** t' **where** t': $\Gamma,\gamma,p\vdash \langle rs_1,\ s\rangle \Rightarrow t'$ $\Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Empty}]\ @\ rs_2,\ t'\rangle \Rightarrow t$
by (*blast elim*: *seqE*)
hence $\Gamma,\gamma,p\vdash \langle \mathit{Rule}\ m\ \mathit{Empty}\ \#\ rs_2,\ t'\rangle \Rightarrow t$

by *simp*
then obtain t'' **where** $\Gamma,\gamma,p\vdash \langle[\mathit{Rule}\ m\ \mathit{Empty}],\ t'\rangle \Rightarrow t''$ $\Gamma,\gamma,p\vdash \langle rs_2,\ t''\rangle \Rightarrow t$
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: $\mathit{matches}\ \gamma\ m\ p \Longrightarrow p\vdash' \langle[\mathit{Rule}\ m\ \mathit{Accept}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalAllow}$ |
drop: $\mathit{matches}\ \gamma\ m\ p \Longrightarrow p\vdash' \langle[\mathit{Rule}\ m\ \mathit{Drop}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalDeny}$ |
reject: $\mathit{matches}\ \gamma\ m\ p \Longrightarrow p\vdash' \langle[\mathit{Rule}\ m\ \mathit{Reject}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Decision}\ \mathit{FinalDeny}$ |
log: $\mathit{matches}\ \gamma\ m\ p \Longrightarrow p\vdash' \langle[\mathit{Rule}\ m\ \mathit{Log}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}$ |
empty: $\mathit{matches}\ \gamma\ m\ p \Longrightarrow p\vdash' \langle[\mathit{Rule}\ m\ \mathit{Empty}],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}$ |
nomatch: $\neg\ \mathit{matches}\ \gamma\ m\ p \Longrightarrow p\vdash' \langle[\mathit{Rule}\ m\ a],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}$ |
decision: $p\vdash' \langle rs,\ \mathit{Decision}\ X\rangle \Rightarrow \mathit{Decision}\ X$ |
seq: $[\![p\vdash' \langle rs_1,\ \mathit{Undecided}\rangle \Rightarrow t;\ p\vdash' \langle rs_2,\ t\rangle \Rightarrow t']\!] \Longrightarrow p\vdash' \langle rs_1 @ rs_2,\ \mathit{Undecided}\rangle \Rightarrow t'$ |
call-return: $[\![\ \mathit{matches}\ \gamma\ m\ p;\ \Gamma\ \mathit{chain} = \mathit{Some}\ (rs_1@[\mathit{Rule}\ m'\ \mathit{Return}]@rs_2);$
$\mathit{matches}\ \gamma\ m'\ p;\ p\vdash' \langle rs_1,\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}\ ]\!] \Longrightarrow$
$p\vdash' \langle[\mathit{Rule}\ m\ (\mathit{Call}\ \mathit{chain})],\ \mathit{Undecided}\rangle \Rightarrow \mathit{Undecided}$ |
call-result: $[\![\ \mathit{matches}\ \gamma\ m\ p;\ p\vdash' \langle \mathit{the}\ (\Gamma\ \mathit{chain}),\ \mathit{Undecided}\rangle \Rightarrow t\ ]\!] \Longrightarrow$
$p\vdash' \langle[\mathit{Rule}\ m\ (\mathit{Call}\ \mathit{chain})],\ \mathit{Undecided}\rangle \Rightarrow t$

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

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

lemma *wf*-Γ *rs* $\Longrightarrow$ *p*⊢′ ⟨*rs*, *s*⟩ ⇒ *t* $\longleftrightarrow$ Γ,γ,*p*⊢ ⟨*rs*, *s*⟩ ⇒ *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 Γ,γ,*p*⊢ ⟨[*Rule m a*], *s*⟩ ⇒ *t* $\longleftrightarrow$ Γ,γ,*p*⊢ ⟨[*Rule m′ a*], *s*⟩ ⇒ *t* (**is** *?l* $\longleftrightarrow$ *?r*)
proof −
 {
  fix *m m′*
  assume Γ,γ,*p*⊢ ⟨[*Rule m a*], *s*⟩ ⇒ *t matches* γ *m p* $\longleftrightarrow$ *matches* γ *m′ p*
  hence Γ,γ,*p*⊢ ⟨[*Rule m′ a*], *s*⟩ ⇒ *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 Γ,γ,*p*⊢ ⟨[*Rule* (*MatchAnd m m′*) *a′*], *Undecided*⟩ ⇒ *t* $\longleftrightarrow$ Γ,γ,*p*⊢ ⟨[*Rule m′ a′*], *Undecided*⟩ ⇒ *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 Γ,γ,*p*⊢ ⟨[*Rule* (*MatchNot m*) *a*], *Undecided*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨[], *Undecided*⟩ ⇒ *t* (**is** *?l* ⟷ *?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 Γ,γ,*p*⊢ ⟨[*Rule* (*MatchAnd* (*MatchNot m*) *m'*) *a*], *Undecided*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨[], *Undecided*⟩ ⇒ *t* (**is** *?l* ⟷ *?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 Γ,γ,*p*⊢ ⟨[*Rule* (*MatchAnd m m'*) *a'*], *s*⟩ ⇒ *t* ⟷ Γ,γ,*p*⊢ ⟨[*Rule m' a'*], *s*⟩ ⇒ *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* ⇒ *'a rule list* ⇒ *'a rule list* **where**
add-match m rs = *map* ($\lambda r.$ *case r of Rule m' a'* ⇒ *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 $\Gamma,\gamma,p\vdash \langle add\text{-}match\ m\ rs,\ s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$ (**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 Γ,γ,p⊢ ⟨add-match (MatchNot m) rs, s⟩ ⇒ t ⟷ Γ,γ,p⊢ ⟨[], s⟩ ⇒ t (is
?l s ⟷ ?r s)
  proof (cases s)
    case Undecided
    have ?l Undecided ⟷ ?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 $\Gamma,\gamma,p\vdash \langle$*add-match m rs*, *Undecided*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle[]$, *Undecided*$\rangle \Rightarrow t$
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*:
$\Gamma,\gamma,p\vdash \langle$*add-match* (*MatchNot* (*MatchNot m*)) *rs*, *s*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle$*add-match m rs*, *s*$\rangle \Rightarrow t$
proof(*induction rs*)
case *Nil*
thus *?case*
unfolding *add-match-def* **by** *simp*
next
case (*Cons r rs*)
thus *?case*
by (*cases r*)
(*metis decision decisionD state.exhaust matches.simps*(*2*) *matches-add-match-simp not-matches-add-match-simp*)
qed

lemma *not-matches-add-matchNot-simp*:
$\neg$ *matches* γ *m p* $\Longrightarrow \Gamma,\gamma,p\vdash \langle$*add-match* (*MatchNot m*) *rs*, *s*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle$*rs*, *s*$\rangle \Rightarrow t$
by (*simp add*: *matches-add-match-simp*)

lemma *iptables-bigstep-add-match-and*:
$\Gamma,\gamma,p\vdash \langle$*add-match m1* (*add-match m2 rs*), *s*$\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle$*add-match* (*MatchAnd m1 m2*) *rs*, *s*$\rangle \Rightarrow t$

```
  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 Γ,γ,p⊢ ⟨Rule (MatchAnd m1 (MatchAnd m2 m)) a # add-match m1
(add-match m2 rs), s⟩ ⇒ t ⟷ Γ,γ,p⊢ ⟨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 *Returns*

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*: $\Gamma,\gamma,p\vdash \langle$*add-missing-ret-unfoldings rs1 rs2*, $s\rangle \Rightarrow t \longleftrightarrow$
$\Gamma,\gamma,p\vdash \langle$(*add-match-MatchAnd-foldr* (*map* (λr. *MatchNot* (*get-match r*)) [$r\leftarrow$*rs1*. *get-action r* = *Return*])) *rs2*, $s\ \rangle \Rightarrow t$
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 Γ,γ,p⊢ ⟨*add-match m1* (*add-match m2 rs*), *s*⟩ ⇒ *t* ⟹ Γ,γ,p⊢ ⟨*add-match m2* (*add-match m1 rs*), *s*⟩ ⇒ *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*: Γ,γ,p⊢ ⟨[*Rule* (*MatchAnd m1* (*MatchAnd m2 m*)) *a*], *s*⟩ ⇒ *ti* **and** *2*: Γ,γ,p⊢ ⟨*add-match m1* (*add-match m2 rs*), *ti*⟩ ⇒ *t*
      apply(*simp add*: *add-match-split-fst*)
      apply(*erule seqE-cons*)
      by *simp*
     from *1 r* **have** *base*: Γ,γ,p⊢ ⟨[*Rule* (*MatchAnd m2* (*MatchAnd m1 m*)) *a*], *s*⟩ ⇒ *ti*
      by (*metis matches.simps*(*1*) *matches-rule-iptables-bigstep*)
     from *2 Cons.IH* **have** *IH*: Γ,γ,p⊢ ⟨*add-match m2* (*add-match m1 rs*), *ti*⟩ ⇒ *t* **by** *simp*
     from *base IH seq′-cons* **have** Γ,γ,p⊢ ⟨*Rule* (*MatchAnd m2* (*MatchAnd m1 m*)) *a* # *add-match m2* (*add-match m1 rs*), *s*⟩ ⇒ *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*: $\Gamma,\gamma,p\vdash$ ⟨*rs*, *Undecided*⟩ $\Rightarrow$ *t* $\Longrightarrow$ $\Gamma,\gamma,p\vdash$ ⟨*process-ret rs*, *Undecided*⟩ $\Rightarrow$ *t*
proof (*induction rs*)
  case (*Cons r rs*)
  show *?case*
    proof (*cases r*)
      case (*Rule m′ a′*)
      show $\Gamma,\gamma,p\vdash$ ⟨*process-ret* (*r* # *rs*), *Undecided*⟩ $\Rightarrow$ *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*:$\Gamma,\gamma,p\vdash$ ⟨[*r*], *Undecided*⟩ $\Rightarrow$ *ti* **and** *2*: $\Gamma,\gamma,p\vdash$ ⟨*rs*, *ti*⟩ $\Rightarrow$ *t* **using** *seqE-cons* **by** *metis*
          thus *?thesis*
            proof(*cases ti*)
            case *Undecided*
              with *Cons.IH 2* **have** *IH*: $\Gamma,\gamma,p\vdash$ ⟨*process-ret rs*, *Undecided*⟩ $\Rightarrow$ *t* **by** *simp*
                from *Undecided 1 Call Rule* **have** $\Gamma,\gamma,p\vdash$ ⟨[*Rule m′* (*Call chain*)], *Undecided*⟩ $\Rightarrow$ *Undecided* **by** *simp*
            with *IH* **have** $\Gamma,\gamma,p\vdash$ ⟨*Rule m′* (*Call chain*) # *process-ret rs*, *Undecided*⟩ $\Rightarrow$ *t* **using** *seq′-cons* **by** *fast*
              thus *?thesis* **using** *Rule Call* **by** *force*
            next
            case (*Decision X*)
              with *1 Rule Call* **have** $\Gamma,\gamma,p\vdash$ ⟨[*Rule m′* (*Call chain*)], *Undecided*⟩ $\Rightarrow$ *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-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-seq nomatchD seq seqE-cons skip*)
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*:
$\Gamma,\gamma,p\vdash$ $\langle$*add-match m* (*add-missing-ret-unfoldings rs1 rs2*), *Undecided*$\rangle$ $\Rightarrow$ *Undecided* =
$\Gamma,\gamma,p\vdash$ $\langle$*add-missing-ret-unfoldings rs1* (*add-match m rs2*), *Undecided*$\rangle$ $\Rightarrow$ *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*)) [$r\leftarrow$*rs1* . (*get-action r*) = *Return*])
apply(*simp-all add*: *add-match-distrib*)
done

Completeness

theorem *unfolding-complete*: $\Gamma,\gamma,p\vdash$ $\langle$*rs*,*s*$\rangle$ $\Rightarrow$ *t* $\Longrightarrow$ $\Gamma,\gamma,p\vdash$ $\langle$*process-call* Γ *rs*,*s*$\rangle$ $\Rightarrow$ *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 $\Gamma,\gamma,p\vdash \langle rs_1,\ Undecided\rangle \Rightarrow Undecided$
by *simp*
hence $\Gamma,\gamma,p\vdash \langle$*process-ret* rs_1, *Undecided*$\rangle \Rightarrow$ *Undecided*
by (*rule iptables-bigstep-process-ret-undecided*)
with *Call-return* **have** $\Gamma,\gamma,p\vdash \langle$*process-ret* rs_1 @ *add-missing-ret-unfoldings* rs_1 (*add-match* (*MatchNot m'*) (*process-ret* rs_2)), *Undecided*$\rangle \Rightarrow$ *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 $\Gamma,\gamma,p\vdash$ ⟨*process-ret rs*, *Undecided*⟩ ⇒ *Undecided*
obtains (*noreturn*) $\Gamma,\gamma,p\vdash$ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided*
| (*return*) rs_1 rs_2 *m* **where** *rs* = rs_1@[*Rule m Return*]@rs_2 $\Gamma,\gamma,p\vdash$ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* *matches* γ *m p*
proof −
have $\Gamma,\gamma,p\vdash$ ⟨*process-ret rs*, *Undecided*⟩ ⇒ *Undecided* ⟹
($\Gamma,\gamma,p\vdash$ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided*) ∨
(∃ rs_1 rs_2 *m*. *rs* = rs_1@[*Rule m Return*]@rs_2 ∧ $\Gamma,\gamma,p\vdash$ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* ∧ *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* ≠ *Return*)
case *True*
with *r Cons.prems* **have** *prems-r*: $\Gamma,\gamma,p\vdash$ ⟨[*Rule m a*], *Undecided*⟩ ⇒ *Undecided* **and** *prems-rs*: $\Gamma,\gamma,p\vdash$ ⟨*process-ret rs*, *Undecided*⟩ ⇒ *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** $\Gamma,\gamma,p\vdash$ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided* ∨ (∃ rs_1 rs_2 *m*. *rs* = rs_1 @ [*Rule m Return*] @ rs_2 ∧ $\Gamma,\gamma,p\vdash$ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* ∧ *matches* γ *m p*) **by** *simp*
thus $\Gamma,\gamma,p\vdash$ ⟨*r* # *rs*, *Undecided*⟩ ⇒ *Undecided* ∨ (∃ rs_1 rs_2 *m*. *r* # *rs* = rs_1 @ [*Rule m Return*] @ rs_2 ∧ $\Gamma,\gamma,p\vdash$ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* ∧ *matches* γ *m p*) (**is** *?goal*)
proof(*elim disjE*)
assume $\Gamma,\gamma,p\vdash$ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided*
hence $\Gamma,\gamma,p\vdash$ ⟨*r* # *rs*, *Undecided*⟩ ⇒ *Undecided* **using** *prems-r* **by** (*metis r seq'-cons*)
thus *?goal* **by** *simp*
next
assume (∃ rs_1 rs_2 *m*. *rs* = rs_1 @ [*Rule m Return*] @ rs_2 ∧ $\Gamma,\gamma,p\vdash$ ⟨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** $\Gamma,\gamma,p\vdash$ ⟨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 ∧ $\Gamma,\gamma,p\vdash$ ⟨rs_1, *Undecided*⟩ ⇒ *Undecided* ∧ *matches* γ *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*: Γ,γ,*p*⊢ ⟨*add-match* (*MatchNot m*) (*process-ret rs*), *Undecided*⟩ ⇒ *Undecided* **by** *simp*
show Γ,γ,*p*⊢ ⟨*r* # *rs*, *Undecided*⟩ ⇒ *Undecided* ∨ (∃ *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*) (**is** *?goal*)
proof(*cases matches* γ *m p*)
case *True*

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*=[] **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*: ⋀*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*:
$\Gamma,\gamma,p\vdash$ ⟨*add-match* (*MatchNot m*) *rs*, *Undecided*⟩ $\Rightarrow$ *Undecided* $\Longrightarrow$ *matches* γ m p $\vee$ $\Gamma,\gamma,p\vdash$ ⟨*rs*, *Undecided*⟩ $\Rightarrow$ *Undecided*
by (*metis matches.simps*(*2*) *matches-add-match-simp*)

lemma *iptables-bigstep-process-ret-DecisionD*: $\Gamma,\gamma,p\vdash$ ⟨*process-ret rs*, *s*⟩ $\Rightarrow$ *Decision X* $\Longrightarrow$ $\Gamma,\gamma,p\vdash$ ⟨*rs*, *s*⟩ $\Rightarrow$ *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*: $\Gamma,\gamma,p\vdash$ ⟨[*Rule m Return*], *Undecided*⟩ $\Rightarrow$ 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*)$,\gamma,p\vdash$ ⟨*rs'*, *s*⟩ $\Rightarrow$ t $\longleftrightarrow$ Γ(*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*)
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 $a = Log \lor a = Empty$
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** $\Gamma=\Gamma(chain \mapsto rs)$] **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** $\Gamma=\Gamma(chain \mapsto rs)$] **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*: $\Gamma(chain \mapsto Rule\ m\ a\ \#\ rs),\gamma,p\vdash \langle Rule\ m\ a\ \#\ rs_1,\ Undecided\rangle \Rightarrow Undecided$
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*: $(\Gamma(chain \mapsto Rule\ m\ a\ \#\ rs))\ chain' = Some\ (rs_1 @ Rule\ m''\ Return\ \#\ rs_2)$
by(*simp*)
with *Call-return* **have** $\Gamma(chain \mapsto Rule\ m\ a\ \#\ rs),\gamma,p\vdash \langle [Rule\ m'\ (Call\ chain')],\ Undecided\rangle \Rightarrow Undecided$
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 $\Gamma(chain \mapsto Rule\ m\ a\ \#\ rs),\gamma,p\vdash \langle [Rule\ m'\ a'],\ Undecided\rangle \Rightarrow Undecided$ **using** *Call-return* **by** *simp*

next
case *True*
with *Call-return* **have** *x*: $(\Gamma(chain \mapsto Rule\ m\ a\ \#\ rs))\ chain' = Some\ (Rule\ m\ a\ \#\ rs_1\ @\ Rule\ m''\ Return\ \#\ rs_2)$
by(*simp*)
with *Call-return* **have** $\Gamma(chain \mapsto Rule\ m\ a\ \#\ rs),\gamma,p\vdash \langle[Rule\ m'\ (Call\ chain')],\ Undecided\rangle \Rightarrow Undecided$
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 $\Gamma(chain \mapsto Rule\ m\ a\ \#\ rs),\gamma,p\vdash \langle[Rule\ m'\ a'],\ Undecided\rangle \Rightarrow Undecided$ **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*: $(\lambda b.\ if\ b = chain\ then\ Some\ rs\ else\ \Gamma\ b) = \Gamma(chain \mapsto rs)$
by *auto*

lemma *no-recursive-calls-helper*:
assumes $\Gamma,\gamma,p\vdash \langle[Rule\ m\ (Call\ chain)],\ Undecided\rangle \Rightarrow t$
and *matches* γ *m p*
and $\Gamma\ chain = Some\ [Rule\ m\ (Call\ chain)]$
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*
⟹ *matches* γ *m p* ⟹ *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 Γ(*chain* $\mapsto$ *rs*),γ,p⊢ ⟨[*Rule m a*], *Undecided*⟩ ⇒ *Undecided*
and Γ(*chain* $\mapsto$ *Rule m a* # *rs*),γ,p⊢ ⟨*rs'*, *s*⟩ ⇒ *t*
shows Γ(*chain* $\mapsto$ *rs*),γ,p⊢ ⟨*rs'*, *s*⟩ ⇒ *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-hlp*)
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 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** Γ=(λb. *if b* = *chain then Some rs else* Γ *b*)])
apply(*case-tac t*)
apply(*simp*)
apply(*frule iptables-bigstep-to-undecided*[**where** Γ=(λb. *if b* = *chain then Some rs else* Γ *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 ¬ (∃ *chain*. *a* = *Call chain*)
shows Γ,γ,p⊢ ⟨[*Rule m a*], *s*⟩ ⇒ *t* ⟷ Γ',γ,p⊢ ⟨[*Rule m a*], *s*⟩ ⇒ *t*
proof −
{
fix Γ Γ'
have Γ,γ,p⊢ ⟨[*Rule m a*], *s*⟩ ⇒ *t* ⟹ Γ',γ,p⊢ ⟨[*Rule m a*], *s*⟩ ⇒ *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 Γ,γ,p⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided* **and** Γ',γ,p⊢ ⟨*rs′*, *Undecided*⟩ ⇒ *Undecided*
shows Γ,γ,p⊢ ⟨[*Rule m* (*Call chain*)], *s*⟩ ⇒ *t* ⟷ Γ',γ,p⊢ ⟨[*Rule m* (*Call chain*)], *s*⟩ ⇒ *t*

proof −
{
fix Γ Γ′ *rs rs′*
assume *assms*:
Γ *chain* = *Some rs* Γ′ *chain* = *Some rs′*
Γ,γ,p⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided* Γ′,γ,p⊢ ⟨*rs′*, *Undecided*⟩ ⇒ *Undecided*
have Γ,γ,p⊢ ⟨[*Rule m* (*Call chain*)], *s*⟩ ⇒ *t* ⟹ Γ′,γ,p⊢ ⟨[*Rule m* (*Call chain*)], *s*⟩ ⇒ *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* ↦ *Rule m* (*Call foo*) # *rs′*),γ,p⊢ ⟨*rs*, *Undecided*⟩ ⇒ *Undecided*
and *matches* γ *m p*
shows Γ(*chain* ↦ *rs′*),γ,p⊢ ⟨*rs*, *Undecided*⟩ ⇒ *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* ↦ *list* @ *Rule m′ Return* # rs_2) **in** *seqE-cons*)
apply(*frule-tac* Γ=Γ(*chain* ↦ *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*: $\Gamma,\gamma,p\vdash \langle$*process-ret* (*add-match* *m* *rs*), *s*$\rangle \Rightarrow$ *t* $\Longrightarrow \Gamma,\gamma,p\vdash \langle$*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*: $\Gamma,\gamma,p\vdash$ ⟨*process-ret* (*add-match m rs*), *s*⟩ $\Rightarrow$ *t* $\longleftrightarrow$ $\Gamma,\gamma,p\vdash$ ⟨*add-match m* (*process-ret rs*), *s*⟩ $\Rightarrow$ *t*
by (*metis process-ret-add-match-dist1 process-ret-add-match-dist2*)

lemma *process-ret-Undecided-sound*:
assumes Γ(*chain* $\mapsto$ *rs*),γ,$p\vdash$ ⟨*process-ret* (*add-match m rs*), *Undecided*⟩ $\Rightarrow$ *Undecided*
shows Γ(*chain* $\mapsto$ *rs*),γ,$p\vdash$ ⟨[*Rule m* (*Call chain*)], *Undecided*⟩ $\Rightarrow$ *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** Γ=Γ(*chain* $\mapsto$ [])]
have (Γ(*chain* $\mapsto$ [])) *chain* = *Some* [] $\Longrightarrow$ Γ(*chain* $\mapsto$ []),γ,$p\vdash$ ⟨[], *Undecided*⟩ $\Rightarrow$ *Undecided* $\Longrightarrow$ Γ(*chain* $\mapsto$ []),γ,$p\vdash$ ⟨[*Rule m* (*Call chain*)], *Undecided*⟩ $\Rightarrow$ *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*: Γ(*chain* $\mapsto$ *Rule* m' a' # *rs*),γ,*p*$\vdash$ $\langle$*process-ret* (*add-match* m (*Rule* m' a' # *rs*)), *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
by *fast*
hence *prems-simplified*: Γ(*chain* $\mapsto$ *Rule* m' a' # *rs*),γ,*p*$\vdash$ $\langle$*process-ret* (*Rule* m' a' # *rs*), *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
using *matches* **by** (*metis matches-add-match-simp process-ret-add-match-dist*)

have Γ(*chain* $\mapsto$ *Rule* m' a' # *rs*),γ,*p*$\vdash$ $\langle$[*Rule* m (*Call chain*)], *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
proof (*cases* a' = *Return*)
case *True*
note a' = *this*
have Γ(*chain* $\mapsto$ *Rule* m' *Return* # *rs*),γ,*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 *?*Γ' = Γ(*chain* $\mapsto$ *Rule m' a'* # *rs*)
from *IH-pre* **show** *?thesis*
proof (*cases rule*: *iptables-bigstep-process-ret-cases3*)
case *noreturn*
with *call* **have** *?*Γ',γ,$p\vdash$ $\langle$*Rule m'* (*Call c*) # *rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
by (*metis step seq-cons*)
from *call* **have** *?*Γ' *chain* = *Some* (*Rule m'* (*Call c*) # *rs*)
by *simp*
from *matches* **show** *?thesis*
by (*rule call-result*) *fact+*
next
case (*return* rs_1 rs_2 *new-m'*)
with *call* **have** *?*Γ' *chain* = *Some* ((*Rule m'* (*Call c*) # rs_1) @ [*Rule new-m' Return*] @ rs_2)
by *simp*
from *call return step* **have** *?*Γ',γ,$p\vdash$ $\langle$*Rule m'* (*Call c*) # rs_1, *Undecided*$\rangle$ $\Rightarrow$ *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 Γ(*chain* $\mapsto$ *rs*),γ,$p\vdash$ $\langle$*process-ret* (*add-match m rs*), *Undecided*$\rangle$ $\Rightarrow$ *Decision X*
shows Γ(*chain* $\mapsto$ *rs*),γ,$p\vdash$ $\langle$[*Rule m* (*Call chain*)], *Undecided*$\rangle$ $\Rightarrow$ *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*: Γ(*chain* $\mapsto$ *Rule m′ a′* # *rs*),γ,$p\vdash$ $\langle$*process-ret* (*add-match m* (*Rule m′ a′* # *rs*)), *Undecided*$\rangle$ $\Rightarrow$ *Decision X*
by *fast*
hence *prems-simplified*: Γ(*chain* $\mapsto$ *Rule m′ a′* # *rs*),γ,$p\vdash$ $\langle$*process-ret* (*Rule m′ a′* # *rs*), *Undecided*$\rangle$ $\Rightarrow$ *Decision X*
using *matches* **by** (*metis matches-add-match-simp process-ret-add-match-dist*)

have Γ(*chain* $\mapsto$ *Rule m′ a′* # *rs*),γ,$p\vdash$ $\langle$[*Rule m* (*Call chain*)], *Undecided*$\rangle$ $\Rightarrow$ *Decision X*
proof (*cases a′* = *Return*)
case *True*
note *a′* = *this*
have Γ(*chain* $\mapsto$ *Rule m′ Return* # *rs*),γ,$p\vdash$ $\langle$[*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** Γ(*chain* $\mapsto$ *rs*),γ,$p\vdash$ $\langle$*process-ret* (*Rule m' a'* # *rs*), *Undecided*$\rangle \Rightarrow$ *Decision X*
by (*metis update-Gamma-nomatch*)
with *a' matches matches'* **have** Γ(*chain* $\mapsto$ *rs*),γ,$p\vdash$ $\langle$*add-match m* (*process-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*: Γ(*chain* $\mapsto$ *Rule m' a'* # *rs*),γ,$p\vdash$ $\langle$[*Rule m' a'*], *Undecided*$\rangle$ $\Rightarrow$ *ti*
and *IH-pre*: Γ(*chain* $\mapsto$ *Rule m' a'* # *rs*),γ,$p\vdash$ $\langle$*process-ret rs*, *ti*$\rangle \Rightarrow$ *Decision X*
by (*auto simp*: *process-ret-split-fst-NeqReturn elim*: *seqE-cons*)

hence Γ(*chain* $\mapsto$ *Rule m' a'* # *rs*),γ,$p\vdash$ $\langle$*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-ret rs* $\Longrightarrow \forall\, r \in$ *set rs*. *get-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*: Γ *chain* = *Some rs*
and *a2*: *matches* γ *m p*
and *a3*: $\forall$ *r*$\in$*set rs*. *get-action r* = *Return*
shows Γ,γ,*p*$\vdash$ $\langle$[*Rule m* (*Call chain*)], *Undecided*$\rangle$ $\Rightarrow$ *Undecided*
proof (*cases* $\exists$ *r* $\in$ *set rs*. *matches* γ (*get-match r*) *p*)
case *True*
hence ($\exists$ *rs1 r rs2*. *rs* = *rs1* @ *r* # *rs2* $\wedge$ *matches* γ (*get-match r*) *p* $\wedge$ ($\forall$ *r*$'$$\in$*set rs1*. $\neg$ *matches* γ (*get-match r*$'$) *p*))
by (*subst split-list-first-prop-iff* [*symmetric*])
then obtain *rs1 r rs2*
where $*$: *rs* = *rs1* @ *r* # *rs2* *matches* γ (*get-match r*) *p* $\forall$ *r*$'$$\in$*set rs1*. $\neg$ *matches* γ (*get-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 Γ,γ,*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 Γ(*chain* $\mapsto$ *rs*),γ,*p*$\vdash$ $\langle$*process-ret* (*add-match m rs*), *Undecided*$\rangle$ $\Rightarrow$ *t*
shows Γ(*chain* $\mapsto$ *rs*),γ,*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-action* (*case r of Rule m*$'$ *x* $\Rightarrow$ *Rule* (*MatchAnd m m*$'$) *x*) = *get-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-chain Γ *rs* $\equiv$ ($\forall$ *r* $\in$ *set rs*. $\forall$ *chain*. *get-action r* = *Call chain* $\longrightarrow$ Γ *chain* $\neq$ *None*)
lemma *wf-chain-append*: *wf-chain* Γ (*rs1*@*rs2*) $\longleftrightarrow$ *wf-chain* Γ *rs1* $\wedge$ *wf-chain* Γ *rs2*
by(*simp add*: *wf-chain-def*, *blast*)
lemma *wf-chain-process-ret*: *wf-chain* Γ *rs* $\Longrightarrow$ *wf-chain* Γ (*process-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-chain* Γ *rs* $\Longrightarrow$ *wf-chain* Γ (*add-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-chain* Γ *rs* $\Longrightarrow$ $\Gamma,\gamma,p\vdash$ $\langle$*process-call* Γ *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** Γ,γ,p⊢ ⟨(*process-call* Γ ^^ *n*) *rs*, *s*⟩ ⇒ *t* = Γ,γ,p⊢ ⟨*rs*, *s*⟩ ⇒ *t* **by** *blast*
from *Suc.prems* **have** ∀ *a*∈*ran* Γ ∪ {*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** ∀ *a*∈*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 *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*) $C = (\forall a.\ disc\ a \longrightarrow C\ (sel\ a) = a)$
declare *wf-disc-sel.simps*[*simp del*]

end
theory *IPSpace-Syntax*
imports *Main String ../Bitmagic/IPv4Addr ../Datatype-Selectors*
begin

4 Primitive Matchers: IP Space Matcher

Primitive Match Conditions which only support IPv4 addresses and layer 4 protocols. Used to partition the IPv4 address space.

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

datatype *ipt-protocol* = *ProtAll* | *ProtTCP* | *ProtUDP*

datatype-new *iptrule-match* =
is-Src: *Src* (*src-range*: *ipt-ipv4range*)
| *is-Dst*: *Dst* (*dst-range*: *ipt-ipv4range*)
| *is-Prot*: *Prot* (*prot-sel*: *ipt-protocol*)

| *is-Extra*: *Extra* (*extra-sel*: *string*)

lemma *wf-disc-sel-iptrule-match*[*simp*]:
wf-disc-sel (*is-Src*, *src-range*) *Src*
wf-disc-sel (*is-Dst*, *dst-range*) *Dst*
wf-disc-sel (*is-Prot*, *prot-sel*) *Prot*
wf-disc-sel (*is-Extra*, *extra-sel*) *Extra*
by(*simp-all add*: *wf-disc-sel.simps*)

4.1 Example Packet

datatype *protPacket* = *ProtTCP* | *ProtUDP*
record *packet* = *src-ip* :: *ipv4addr*
dst-ip :: *ipv4addr*
prot :: *protPacket*

hide-const (**open**) *ProtTCP ProtUDP*

fun *ipv4s-to-set* :: *ipt-ipv4range* ⇒ *ipv4addr set* **where**
ipv4s-to-set (*Ip4AddrNetmask base m*) = *ipv4range-set-from-bitmask* (*ipv4addr-of-dotteddecimal base*) *m* |
ipv4s-to-set (*Ip4Addr ip*) = { *ipv4addr-of-dotteddecimal 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*: *addr* ∈ *ipv4s-to-set X* = (
case X of (*Ip4AddrNetmask pre len*) ⇒ ((*ipv4addr-of-dotteddecimal pre*) *AND* ((*mask len*) << (*32* − *len*))) ≤ *addr* ∧ *addr* ≤ (*ipv4addr-of-dotteddecimal pre*) *OR* (*mask* (*32* − *len*))
| *Ip4Addr ip* ⇒ (*addr* = (*ipv4addr-of-dotteddecimal ip*)))
apply(*cases X*)
apply(*simp*)
apply(*simp add*: *ipv4range-set-from-bitmask-alt*)
done

— Misc

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

end
theory *Example-Semantics*
imports ../*Call-Return-Unfolding* ../*Primitive-Matchers*/*IPSpace-Syntax*
begin

5 Examples Big Step Semantics

we use a primitive matcher which always apllies.

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*)))
definition *p*=(|*src-ip*=*0*, *dst-ip*=*0*, *prot*=*protPacket.ProtTCP*|)
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 ((*λrs. process-call* [*''FORWARD''* ↦ [*Rule m* (*Call ''foo''*), *Rule m Drop*], *''foo''* ↦ []] *rs*)^^*2*)
[*Rule MatchAny* (*Call ''FORWARD''*)]
= [*Rule* (*MatchAnd MatchAny m*) *Drop*] **by** *eval*

hide-const *m p*

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* :: (*iptrule-match*, *packet*) *matcher* **where**
applies-exampleMatchExact (*Src* (*Ip4Addr addr*)) *p* ⟷ *src-ip p* = (*ipv4addr-of-dotteddecimal addr*) |
applies-exampleMatchExact (*Dst* (*Ip4Addr addr*)) *p* ⟷ *dst-ip p* = (*ipv4addr-of-dotteddecimal addr*) |
applies-exampleMatchExact (*Prot ProtAll*) *p* ⟷ *True* |
applies-exampleMatchExact (*Prot ipt-protocol.ProtTCP*) *p* ⟷ *prot p* = *protPacket.ProtTCP* |
applies-exampleMatchExact (*Prot ipt-protocol.ProtUDP*) *p* ⟷ *prot p* = *protPacket.ProtUDP*

lemma [*''FORWARD''* ↦ [*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 ipt-protocol.ProtTCP*)) *Accept*,
Rule (*Match* (*Prot ipt-protocol.ProtTCP*)) *Drop*]
],*applies-exampleMatchExact*,⦇*src-ip*=(*ipv4addr-of-dotteddecimal* (*1*,*2*,*3*,*4*)), *dst-ip*=(*ipv4addr-of-dotteddecimal* (*0*,*0*,*0*,*0*)), *prot*=*protPacket.ProtTCP*⦈⊢
⟨[*Rule MatchAny* (*Call ''FORWARD''*)], *Undecided*⟩ ⇒ (*Decision FinalAllow*)
apply(*rule call-result*)
apply(*auto*)

apply(*rule Semantics.seq-cons*)
apply(*auto intro*: *Semantics.nomatch simp add*: *ipv4addr-of-dotteddecimal.simps ipv4addr-of-nat-def*)
apply(*rule Semantics.seq-cons*)
apply(*auto intro*: *Semantics.log simp add*: *ipv4addr-of-dotteddecimal.simps ipv4addr-of-nat-def*)
apply(*rule Semantics.seq-cons*)
apply(*auto intro*: *Semantics.accept*)
apply(*auto intro*: *Semantics.decision*)
done

end
theory *Ternary*
imports *Main*
begin

6 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* $\Longrightarrow$ *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*

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

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

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*)

lemma *eval-ternary-simps-2*: *eval-ternary-And* (*bool-to-ternary P*) *T* = *TernaryTrue* $\longleftrightarrow$ *P* $\wedge$ *T* = *TernaryTrue*
eval-ternary-And T (*bool-to-ternary P*) = *TernaryTrue* $\longleftrightarrow$ *P* $\wedge$ *T* = *TernaryTrue*
apply(*case-tac* [!] *P*)
apply(*simp-all add*: *eval-ternary-simps-simple*)
done

lemma *eval-ternary-simps-3*: *eval-ternary-And* (*ternary-ternary-eval x*) *T* = *TernaryTrue* $\longleftrightarrow$ (*ternary-ternary-eval x* = *TernaryTrue*) $\wedge$ (*T* = *TernaryTrue*)
eval-ternary-And T (*ternary-ternary-eval x*) = *TernaryTrue* $\longleftrightarrow$ (*ternary-ternary-eval x* = *TernaryTrue*) $\wedge$ (*T* = *TernaryTrue*)
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*

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

6.1 Negation Normal Form

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

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

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

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

NNF-ternary (*TernaryNot* (*TernaryOr t1 t2*)) = *TernaryAnd* (*NNF-ternary* (*TernaryNot t1*)) (*NNF-ternary* (*TernaryNot t2*))

lemma *NNF-ternary-correct*: *ternary-ternary-eval* (*NNF-ternary t*) = *ternary-ternary-eval t*
apply(*induction t rule*: *NNF-ternary.induct*)
apply(*simp-all add*: *eval-ternary-DeMorgan eval-ternary-idempotence-Not*)
done

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

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

7 Packet Matching in Ternary Logic

The matcher for a primitive match expression *'a*

type-synonym (*'a*, *'packet*) *exact-match-tac*=*'a* ⇒ *'packet* ⇒ *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* ⇒ *'packet* ⇒ *bool*

type-synonym (*'a*, *'packet*) *match-tac*=((*'a*, *'packet*) *exact-match-tac* × *'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* ⇒ *'packet* ⇒ *'a match-expr* ⇒ *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*

the *ternaryformulas* we construct never have Or expressions.

fun *ternary-has-or* :: *ternaryformula* ⇒ *bool* **where**
ternary-has-or (*TernaryOr* - -) ⟷ *True* |
ternary-has-or (*TernaryAnd t1 t2*) ⟷ *ternary-has-or t1* ∨ *ternary-has-or t2* |
ternary-has-or (*TernaryNot t*) ⟷ *ternary-has-or t* |
ternary-has-or (*TernaryValue* -) ⟷ *False*
lemma *map-match-tac--does-not-use-TernaryOr*: ¬ (*ternary-has-or* (*map-match-tac* *β p m*))
by(*induction m*, *simp-all*)

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 (β, α) *m a p* ⟹
(*ternary-ternary-eval* (*map-match-tac* β *p m*) = *TernaryUnknown* ⟹ α *a p* ⟹ *P*) ⟹
(*ternary-ternary-eval* (*map-match-tac* β *p m*) = *TernaryTrue* ⟹ *P*)
⟹ *P*
apply(*induction m*)
apply(*auto split*: *option.split-asm simp*: *matches-case-tuple ternary-eval-def ternary-to-bool-bool-to-ternary elim*: *ternary-to-bool.elims*)
done

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

lemma ⟦ *ternary-ternary-eval* (*map-match-tac* β *p expr*) = *TernaryUnknown* ⟧ ⟹ *matches* (β, α) *expr a p* ⟷ *matches* (β, α) (*MatchNot expr*) *a p*
by(*simp add*: *matches-case-ternaryvalue-tuple*)

lemma *bunch-of-lemmata-about-matches*:
matches γ (*MatchAnd m1 m2*) *a p* ⟷ *matches* γ *m1 a p* ∧ *matches* γ *m2 a p*
matches γ *MatchAny a p*
matches γ (*MatchNot MatchAny*) *a p* ⟷ *False*
matches (β, α) (*Match expr*) *a p* = (*case ternary-to-bool* (β *expr p*) *of Some r* ⇒ *r* | *None* ⇒ (α *a p*))
matches (β, α) (*Match expr*) *a p* = (*case* (β *expr p*) *of TernaryTrue* ⇒ *True* | *TernaryFalse* ⇒ *False* | *TernaryUnknown* ⇒ (α *a p*))
matches γ (*MatchNot* (*MatchNot m*)) *a p* ⟷ *matches* γ *m a p*
apply(*case-tac* [!] γ)
by (*simp-all split*: *ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)

lemma *matches-DeMorgan*: *matches* γ (*MatchNot* (*MatchAnd m1 m2*)) *a p* ⟷ (*matches* γ (*MatchNot m1*) *a p*) ∨ (*matches* γ (*MatchNot m2*) *a p*)
by (*cases* γ) (*simp split*: *ternaryvalue.split add*: *matches-case-ternaryvalue-tuple eval-ternary-DeMorgan*)

7.1 Ternary Matcher Algebra

lemma *matches-and-comm*: *matches* γ (*MatchAnd m m'*) *a p* ⟷ *matches* γ (*MatchAnd m' m*) *a p*
apply(*cases* γ, *rename-tac* β α, *clarify*)
apply(*simp split*: *ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)

by (*metis eval-ternary-And-comm ternaryvalue.distinct*(*1*) *ternaryvalue.distinct*(*3*) *ternaryvalue.distinct*(*5*))

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

lemma (*TernaryNot* (*map-match-tac* β *p* (*m*))) = (*map-match-tac* β *p* (*MatchNot* *m*))
by (*metis map-match-tac.simps*(*2*))

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

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*)

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*))
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*))

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
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*

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*
 apply(*simp split*: *ternaryvalue.split-asm ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)
 done

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*
 apply(*simp split*: *ternaryvalue.split-asm ternaryvalue.split add*: *matches-case-ternaryvalue-tuple*)
 done

Optimize away MatchAny matches

fun *opt-MatchAny-match-expr* :: *'a match-expr* $\Rightarrow$ *'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*) = *m* |
 opt-MatchAny-match-expr (*MatchAnd m MatchAny*) = *m* |
 opt-MatchAny-match-expr (*MatchAnd m* (*MatchNot MatchAny*)) = (*MatchNot MatchAny*) |
 opt-MatchAny-match-expr (*MatchAnd* (*MatchNot MatchAny*) *m*) = (*MatchNot MatchAny*) |
 opt-MatchAny-match-expr (*MatchAnd m1 m2*) = *MatchAnd* (*opt-MatchAny-match-expr m1*) (*opt-MatchAny-match-expr m2*)

need to apply multiple times until it stabelizes

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

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

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

lemma *wf-unknown-match-tacD-False1*: *wf-unknown-match-tac* $\alpha \Longrightarrow \neg$ *matches* (β, α) *m Reject p* $\Longrightarrow$ *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

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

lemma *bool-to-ternary-simp1*: *bool-to-ternary X* = *TernaryTrue* $\longleftrightarrow$ *X*
by (*metis bool-to-ternary.elims ternaryvalue.distinct*(*1*))
lemma *bool-to-ternary-simp2*: *bool-to-ternary Y* = *TernaryFalse* $\longleftrightarrow \neg$ *Y*
by (*metis bool-to-ternary.elims ternaryvalue.distinct*(*1*))
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*)
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*)
lemma *bool-to-ternary-simp5*: $\neg$ *eval-ternary-Not* (*bool-to-ternary X*) = *TernaryUnknown*
by (*metis bool-to-ternary-Unknown eval-ternary-Not-UnknownD*)
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*
hide-fact *bool-to-ternary-simp1 bool-to-ternary-simp2 bool-to-ternary-simp3 bool-to-ternary-simp4 bool-to-ternary-simp5*

7.2 Removing Unknown Primitives

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 ($\forall$ *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*

(∀ *p*. *ternary-ternary-eval* (*map-match-tac* β *p* (*Match A*)) = *TernaryUnknown*)
then
if (∀ *p*. α *a p*) *then MatchAny else if* (∀ *p*. ¬ α *a p*) *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*) |

— ¬ (*a* ∧ *b*) = ¬ *b* ∨ ¬ *a* and ¬ *Unknown* = *Unknown*
remove-unknowns-generic (β, α) *a* (*MatchNot* (*MatchAnd m1 m2*)) =
(*if* (*remove-unknowns-generic* (β, α) *a* (*MatchNot m1*)) = *MatchAny* ∨
(*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* ∨ *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′*)))
)
apply(*cases* γ)
apply(*simp*)
done

lemma *a* = *Accept* ∨ *a* = *Drop* ⟹ *matches* (β, α) (*remove-unknowns-generic* (β, α) *a* (*MatchNot* (*Match A*))) *a p* = *matches* (β, α) (*MatchNot* (*Match A*)) *a p*
apply(*simp*)
apply(*simp add*: *bunch-of-lemmata-about-matches matches-case-ternaryvalue-tuple*)
by *presburger*

lemma $a = \mathit{Accept} \lor a = \mathit{Drop} \Longrightarrow \gamma = (\beta, \alpha) \Longrightarrow$
matches (β, α) (*remove-unknowns-generic* γ *a m*) *a* =
matches (β, α) *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*)[*1*]
apply(*simp add*: *matches-case-ternaryvalue-tuple*)
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

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

8 Embedded Ternary-Matching Big Step Semantics

lemma *rules-singleton-rev-E*: [*Rule m a*] = rs_1 @ rs_2 $\Longrightarrow$ (rs_1 = [*Rule m a*] $\Longrightarrow$ rs_2 = [] $\Longrightarrow$ *P m a*) $\Longrightarrow$ (rs_1 = [] $\Longrightarrow$ rs_2 = [*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*
(-,-$\vdash$ $\langle$-, -$\rangle$ $\Rightarrow_\alpha$ - [*60*,*60*,*20*,*98*,*98*] *89*)
for γ **and** *p* **where**
skip: γ,*p*$\vdash$ $\langle$[], *t*$\rangle$ $\Rightarrow_\alpha$ *t* |
accept: $[\![$*matches* γ *m Accept p*$]\!]$ $\Longrightarrow$ γ,*p*$\vdash$ $\langle$[*Rule m Accept*], *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalAllow* |
drop: $[\![$*matches* γ *m Drop p*$]\!]$ $\Longrightarrow$ γ,*p*$\vdash$ $\langle$[*Rule m Drop*], *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalDeny* |
reject: $[\![$*matches* γ *m Reject p*$]\!]$ $\Longrightarrow$ γ,*p*$\vdash$ $\langle$[*Rule m Reject*], *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalDeny* |
log: $[\![$*matches* γ *m Log p*$]\!]$ $\Longrightarrow$ γ,*p*$\vdash$ $\langle$[*Rule m Log*], *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Undecided* |
empty: $[\![$*matches* γ *m Empty p*$]\!]$ $\Longrightarrow$ γ,*p*$\vdash$ $\langle$[*Rule m Empty*], *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Undecided* |
nomatch: $[\![\neg$ *matches* γ *m a p*$]\!]$ $\Longrightarrow$ γ,*p*$\vdash$ $\langle$[*Rule m a*], *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Undecided* |
decision: γ,*p*$\vdash$ $\langle$*rs*, *Decision X*$\rangle$ $\Rightarrow_\alpha$ *Decision X* |
seq: $[\![\gamma$,*p*$\vdash$ $\langle rs_1$, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *t*; γ,*p*$\vdash$ $\langle rs_2$, *t*$\rangle$ $\Rightarrow_\alpha$ $t'$$]\!]$ $\Longrightarrow$ γ,*p*$\vdash$ $\langle rs_1$@rs_2, *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,$p\vdash$ $\langle rs,s\rangle \Rightarrow_\alpha t \Longrightarrow$
$(\bigwedge t.\ P\ []\ t\ t) \Longrightarrow$
$(\bigwedge m\ a.\ matches\ \gamma\ m\ a\ p \Longrightarrow a = Accept \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalAllow)) \Longrightarrow$
$(\bigwedge m\ a.\ matches\ \gamma\ m\ a\ p \Longrightarrow a = Drop \vee a = Reject \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ (Decision\ FinalDeny)) \Longrightarrow$
$(\bigwedge m\ a.\ matches\ \gamma\ m\ a\ p \Longrightarrow a = Log \vee a = Empty \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided) \Longrightarrow$
$(\bigwedge m\ a.\ \neg\ matches\ \gamma\ m\ a\ p \Longrightarrow P\ [Rule\ m\ a]\ Undecided\ Undecided) \Longrightarrow$
$(\bigwedge rs\ X.\ P\ rs\ (Decision\ X)\ (Decision\ X)) \Longrightarrow$
$(\bigwedge rs\ rs_1\ rs_2\ t\ t'.\ rs = rs_1\ @\ rs_2 \Longrightarrow \gamma,p\vdash \langle rs_1,Undecided\rangle \Rightarrow_\alpha t \Longrightarrow P\ rs_1\ Undecided\ t \Longrightarrow \gamma,p\vdash \langle rs_2,t\rangle \Rightarrow_\alpha t' \Longrightarrow P\ rs_2\ t\ t' \Longrightarrow P\ rs\ Undecided\ t')$
$\Longrightarrow P\ rs\ s\ t$
by (*induction rule*: *approximating-bigstep.induct*) (*simp-all*)

lemma *skipD*: $\gamma,p\vdash \langle [],\ s\rangle \Rightarrow_\alpha t \Longrightarrow s = t$
by (*induction* []::'*a rule list s t rule*: *approximating-bigstep-induct*) (*simp-all*)

lemma *decisionD*: $\gamma,p\vdash \langle rs,\ Decision\ X\rangle \Rightarrow_\alpha t \Longrightarrow t = Decision\ X$
by (*induction rs Decision X t rule*: *approximating-bigstep-induct*) (*simp-all*)

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

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

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

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

lemma *emptyD*: $\gamma,p\vdash \langle [Rule\ m\ Empty],\ Undecided\rangle \Rightarrow_\alpha t \Longrightarrow t = Undecided$

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*: γ,*p*⊢ ⟨*take* (*length rsa*) rs_1, *Undecided*⟩ $\Rightarrow_\alpha$ *t*
by *simp*
from *Seq longer* **obtain** *t2*
where *t2a*: γ,*p*⊢ ⟨*drop* (*length rsa*) rs_1,*t*⟩ $\Rightarrow_\alpha$ *t2*
and *rs2-t2*: γ,*p*⊢ ⟨rs_2,*t2*⟩ $\Rightarrow_\alpha$ *t'*
by *blast*
with *t1 rs2-t2* **have** γ,*p*⊢ ⟨*take* (*length rsa*) rs_1 @ *drop* (*length rsa*) rs_1,*Undecided*⟩ $\Rightarrow_\alpha$ *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*: γ,*p*⊢ ⟨rs_1,*Undecided*⟩ $\Rightarrow_\alpha$ *t1*
and *t1b*: γ,*p*⊢ ⟨*take* (*length rsa* − *length* rs_1) rs_2,*t1*⟩ $\Rightarrow_\alpha$ *t*
by *blast*
from *rsb' Seq.hyps* **have** *t2*: γ,*p*⊢ ⟨*drop* (*length rsa* − *length* rs_1) rs_2,*t*⟩ $\Rightarrow_\alpha$ *t'*
by *blast*
with *seq' t1b* **have** γ,*p*⊢ ⟨rs_2,*t1*⟩ $\Rightarrow_\alpha$ *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 γ,*p*⊢ ⟨*r*#*rs*, *s*⟩ $\Rightarrow_\alpha$ *t*
obtains *t'* **where** γ,*p*⊢ ⟨[*r*],*s*⟩ $\Rightarrow_\alpha$ *t'* γ,*p*⊢ ⟨*rs*,*t'*⟩ $\Rightarrow_\alpha$ *t*
using *assms seq-split* **by** (*metis append-Cons append-Nil*)

lemma *seq-fst*: γ,*p*⊢ ⟨[*r*], *s*⟩ $\Rightarrow_\alpha$ *t* ⟹ γ,*p*⊢ ⟨*rs*, *t*⟩ $\Rightarrow_\alpha$ *t'* ⟹ γ,*p*⊢ ⟨*r* # *rs*, *s*⟩

$\Rightarrow_\alpha t'$
apply(*cases s*)
apply(*simp*)
using *seq* **apply** *fastforce*
apply(*simp*)
apply(*drule decisionD*)
apply(*simp*)
apply(*drule decisionD*)
apply(*simp*)
using *decision* **by** *fast*

fun *approximating-bigstep-fun* :: $('a, 'p)$ *match-tac* $\Rightarrow 'p \Rightarrow 'a$ *rule list* $\Rightarrow$ *state* $\Rightarrow$ *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*
$\neg$ *matches* γ *m a p*
then
approximating-bigstep-fun γ *p rs Undecided*
else
case a of Accept $\Rightarrow$ *Decision FinalAllow*
| *Drop* $\Rightarrow$ *Decision FinalDeny*
| *Reject* $\Rightarrow$ *Decision FinalDeny*
| *Log* $\Rightarrow$ *approximating-bigstep-fun* γ *p rs Undecided*
| *Empty* $\Rightarrow$ *approximating-bigstep-fun* γ *p rs Undecided*
(*$*$unhalndled cases$*$*)
)

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

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* 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* = *Log* $\Longrightarrow$ *P* γ *p rs Undecided*) $\Longrightarrow$ (*a* = *Empty* $\Longrightarrow$ *P* γ *p rs Undecided*) $\Longrightarrow$ *P* γ *p* (*Rule m a* # *rs*) *Undecided*) $\Longrightarrow$
P γ *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*)

8.1 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 γ *p rs s*
apply(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)
apply *blast*
apply *blast*
apply(*auto dest*:*wf-rulesetD*)[*1*]
apply(*frule wf-rulesetD*(*1*), *drule wf-rulesetD*(*2*))
apply(*simp*)
apply(*case-tac a*)
apply(*simp-all*)
apply(*auto simp add*: *wf-ruleset-def*)
done

8.1.1 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) *Undecided* = *approximating-bigstep-fun* γ *p* rs_2 (*approximating-bigstep-fun* γ *p* rs_1 *Undecided*)
apply(*induction* rs_1 *arbitrary*:)
apply *simp-all*
apply(*rename-tac r rs1*)
apply(*case-tac r*, *rename-tac x1 x2*)
apply(*clarify*)
apply(*case-tac* ¬ *matches* γ *x1 x2 p*)
apply(*simp add*: *wf-ruleset-def*)
apply(*simp add*: *wf-ruleset-def*)
apply(*case-tac x2*)
apply *simp-all*
apply(*simp-all add*: *Decision-approximating-bigstep-fun*)
apply *auto*
done

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*
apply(*induction rs1 arbitrary*:)
apply(*simp add*: *wf-ruleset-def*)
apply(*rename-tac r rs1*)
apply(*case-tac r*, *rename-tac x1 x2*)
apply(*clarify*)
apply(*case-tac* ¬ *matches* γ *x1 x2 p*)
apply(*simp add*: *wf-ruleset-def*)
apply(*simp add*: *wf-ruleset-def*)
apply(*case-tac x2*)

apply *simp-all*
apply *auto*
done

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*
apply(*induction rs1 arbitrary*:)
apply *simp-all*
apply(*case-tac t*)
apply(*simp-all add*: *Decision-approximating-bigstep-fun*)

apply(*rename-tac r rs1*)
apply(*case-tac r*, *rename-tac x1 x2*)
apply(*clarify*)
apply(*case-tac* $\neg$ *matches* γ *x1 x2 p*)
apply(*simp add*: *wf-ruleset-def*)
apply(*simp add*: *wf-ruleset-def*)
apply(*case-tac x2*)
apply *simp-all*
apply *auto*
done

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(*case-tac s*)
prefer *2*
apply(*simp add*: *Decision-approximating-bigstep-fun*)
apply(*simp*)
apply(*thin-tac s* = *?un*)
apply(*induction* γ *p rsA Undecided rule*: *approximating-bigstep-fun-induct-wf*)
apply(*simp-all*)
apply (*metis approximating-bigstep-fun-seq-wf*)
apply (*metis Decision-approximating-bigstep-fun approximating-bigstep-fun-seq-wf*)+
done

lemma *approximating-bigstep-fun-singleton-prepend*: *approximating-bigstep-fun* γ *p rsB s* = *approximating-bigstep-fun* γ *p rsC s* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*r*#*rsB*) *s* = *approximating-bigstep-fun* γ *p* (*r*#*rsC*) *s*
apply(*case-tac s*)
prefer *2*

apply(*simp add*: *Decision-approximating-bigstep-fun*)
apply(*simp*)
apply(*cases r*)
apply(*simp split*: *action.split*)
done

8.2 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* $\equiv$ ($\forall\, r\in$*set rs*. (*case get-action r of Call chain* $\Rightarrow$ *False* | *Return* $\Rightarrow$ *False* | *Unknown* $\Rightarrow$ *False* | - $\Rightarrow$ *True*))
apply(*induction rs*)
apply(*simp add*: *good-ruleset-def*)
apply(*simp add*: *good-ruleset-def*)
apply(*thin-tac ?x = ?y*)
apply(*rename-tac r rs*)
apply(*case-tac get-action r*)
apply(*simp-all*)
done

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*)
apply(*simp add*: *good-ruleset-def*)
apply(*rule iffI*)
apply(*clarify*)
apply(*case-tac get-action r*)
apply(*simp-all*)
apply(*clarify*)

apply(*case-tac get-action r*)
apply(*simp-all*)
apply(*fastforce*)+
done

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*: $[\![\ \gamma,p\vdash \langle rs_1,\ s\rangle \Rightarrow_\alpha t\ ]\!] \Longrightarrow$
approximating-bigstep-fun γ p $(rs_1$ @ $rs_2)$ s = *approximating-bigstep-fun* γ p rs_2 t
apply(*induction* rs_1 *s t arbitrary*: rs_2 *rule*: *approximating-bigstep.induct*)
apply(*simp-all add*: *Decision-approximating-bigstep-fun*)
done

lemma *approximating-semantics-imp-fun*: $\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t \Longrightarrow$ *approximating-bigstep-fun* γ *p rs s* = *t*
apply(*induction rs s t rule*: *approximating-bigstep-induct*)
apply(*auto*)[*7*]
apply(*case-tac rs*)
apply(*simp-all*)
apply(*simp add*: *approximating-bigstep-fun-seq-semantics*)
done

lemma *approximating-fun-imp-semantics*: **assumes** *wf-ruleset* γ *p rs*
  shows *approximating-bigstep-fun* γ *p rs s* = *t* $\Longrightarrow$ γ,*p*$\vdash$ $\langle$*rs*, *s*$\rangle$ $\Rightarrow_\alpha$ *t*
 using *assms* **proof**(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct-wf*)
  case (*Empty* γ *p s*)
   thus γ,*p*$\vdash$ $\langle$[], *s*$\rangle$ $\Rightarrow_\alpha$ *t* **using** *skip* **by**(*simp*)
  next
  case (*Decision* γ *p r rs X*)
   hence *t* = *Decision X* **by** *simp*
   thus γ,*p*$\vdash$ $\langle$*r* # *rs*, *Decision X*$\rangle$ $\Rightarrow_\alpha$ *t* **using** *decision* **by** *fast*
  next
  case (*Nomatch* γ *p m a rs*)
   thus γ,*p*$\vdash$ $\langle$*Rule m a* # *rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *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* $\Longrightarrow$

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

lemma *approximating-bigstep-deterministic*: $[\![\ \gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t;\ \gamma,p\vdash \langle rs,\ s\rangle \Rightarrow_\alpha t'\ ]\!] \Longrightarrow t = t'$
apply(*induction arbitrary*: t' *rule*: *approximating-bigstep-induct*)
apply(*auto dest*: *approximating-bigstepD*)[*6*]
by (*metis* (*hide-lams*, *mono-tags*) *append-Nil2 approximating-bigstep-fun.simps*(*1*) *approximating-bigstep-fun-seq-semantics*)

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

fun *rm-LogEmpty* :: *'a rule list* ⇒ *'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*
apply(*induction rs*)
apply(*simp-all*)
apply(*rename-tac r rs*)
apply(*case-tac r*)
apply(*rename-tac m a*)
apply(*simp*)
apply(*case-tac a*)
apply(*simp-all*)
apply(*case-tac* [!] *s*)
apply(*simp-all*)
apply (*metis Decision-approximating-bigstep-fun*)
by (*metis Decision-approximating-bigstep-fun*)

lemma *rm-LogEmpty-seq*: *rm-LogEmpty* (*rs1*@*rs2*) = *rm-LogEmpty rs1* @ *rm-LogEmpty rs2*
apply(*induction rs1*)
apply(*simp-all*)
apply(*case-tac a*)
apply(*simp-all*)
apply(*case-tac x2*)
apply(*simp-all*)
done

lemma *rm-LogEmpty-semantics*: $\gamma,p\vdash \langle$*rm-LogEmpty rs*, $s\rangle \Rightarrow_\alpha t \longleftrightarrow \gamma,p\vdash \langle$*rs*, $s\rangle \Rightarrow_\alpha t$
apply(*rule iffI*)

apply(*induction rs arbitrary*: *s t*)
apply(*simp-all*)
apply(*case-tac a*)
apply(*simp*)
apply(*case-tac x2*)
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(*case-tac a*)
apply(*auto intro*: *approximating-bigstep.intros*)
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*
apply(*induction rs*)
apply(*simp-all add*: *good-ruleset-def simple-ruleset-def*)
apply(*clarify*)
apply(*case-tac a*)
apply(*simp*)
apply(*case-tac x2*)
apply(*simp-all*)
apply *fastforce+*
done

Rewrite *Reject* actions to *Drop* actions

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

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*)
apply(*induction rs*)
apply(*simp-all*)
apply(*rename-tac r rs*)
apply(*case-tac r*)
apply(*rename-tac m a*)
apply(*simp*)
apply(*case-tac a*)
apply(*simp-all*)

apply(*case-tac* [!] *s*)
apply(*simp-all*)
apply(*auto dest*: *wf-unknown-match-tacD-False1 wf-unknown-match-tacD-False2*)
done

lemma *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* $\Rightarrow$ $'a$ *match-expr*) $\Rightarrow$ $'a$ *rule list* $\Rightarrow$ $'a$ *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*
apply(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)
apply(*simp add*: *optimize-matches-def*)
apply(*simp add*: *optimize-matches-def*)
apply(*simp add*: *optimize-matches-def*)
apply(*simp add*: *optimize-matches-def*)
apply(*case-tac a*)
apply(*simp-all*)
done

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* ⇒ *'a match-expr* ⇒ *'a match-expr*) ⇒ *'a rule list* ⇒ *'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*: ∀ *a m*. *matches* γ *m a* = *matches* γ (*f a m*) *a* ⟹ *approximating-bigstep-fun* γ *p* (*optimize-matches-a f rs*) *s* = *approximating-bigstep-fun* γ *p rs s*
apply(*induction* γ *p rs s rule*: *approximating-bigstep-fun-induct*)
apply(*simp add*: *optimize-matches-a-def*)
apply(*simp add*: *optimize-matches-a-def*)
apply(*simp add*: *optimize-matches-a-def*)
apply(*simp add*: *optimize-matches-a-def*)
apply(*case-tac a*)
apply(*simp-all*)
done

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

9 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

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

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

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

10 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* (β, α) *m a p* $\neq$ *Semantics.matches* (λ *atm p*. *case* β *atm p of TernaryTrue* $\Rightarrow$ *True* | *TernaryFalse* $\Rightarrow$ *False* | *TernaryUnknown* $\Rightarrow$ α *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* β *p m*) $\neq$ *TernaryUnknown* $\Longrightarrow$
matcher-agree-on-exact-matches γ β $\Longrightarrow$
Semantics.matches γ *m p* = *Matching-Ternary.matches* (β, α) *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* γ β $\Longrightarrow$
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* γ β $\Longrightarrow$ *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* γ β $\Longrightarrow$
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 β $\Longrightarrow$ *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} γ $\equiv$ (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*
begin

11 Semantics Embedding

11.1 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 $\Longrightarrow$
Γ,γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided* $\Longrightarrow$
(β, *in-doubt-allow*),*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Undecided* $\vee$ (β, *in-doubt-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 $\beta \Longrightarrow$
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* $\Longrightarrow$ *?x* $\Longrightarrow$ *?y*)
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 $\Longrightarrow$
$(\beta,\ in\text{-}doubt\text{-}allow),p\vdash \langle rs,\ Undecided\rangle \Rightarrow_\alpha Undecided \Longrightarrow \Gamma,\gamma,p\vdash \langle rs,\ Un\text{-}$

decided⟩ ⇒ *Undecided* ∨ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *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 ⟹
good-ruleset rs ⟹
(β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny* ⟹ Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *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 ⟹ *good-ruleset rs* ⟹
{*p*. (β, *in-doubt-allow*),*p*⊢ ⟨*rs*, *Undecided*⟩ $\Rightarrow_\alpha$ *Decision FinalDeny*} ⊆ {*p*. Γ,γ,*p*⊢ ⟨*rs*, *Undecided*⟩ ⇒ *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*) $\Longrightarrow$ $\Gamma,\gamma,p\vdash$ $\langle$*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

11.2 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 $\Longrightarrow$
$\Gamma,\gamma,p\vdash$ $\langle$*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 $\Longrightarrow$
$\Gamma,\gamma,p\vdash$ $\langle$*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* $\Longrightarrow$ *?x*)
apply (*metis approximating-bigstep.decision approximating-bigstep.seq Semantics.decisionD state.inject*)
apply(*thin-tac False* $\Longrightarrow$ *?x* $\Longrightarrow$ *?y*)
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 β $\Longrightarrow$
good-ruleset rs $\Longrightarrow$
(β, *in-doubt-deny*),*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Undecided* $\Longrightarrow$ Γ,γ,*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow$ *Undecided* $\vee$ Γ,γ,*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 β $\Longrightarrow$
good-ruleset rs $\Longrightarrow$
(β, *in-doubt-deny*),*p*$\vdash$ $\langle$*rs*, *Undecided*$\rangle$ $\Rightarrow_\alpha$ *Decision FinalAllow* $\Longrightarrow$ Γ,γ,*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* $\Longrightarrow$
$\{p.$ (β, *in-doubt-deny*),$p\vdash \langle rs,$ *Undecided*$\rangle \Rightarrow_\alpha$ *Decision FinalAllow*$\} \subseteq \{p.$ $\Gamma,\gamma,p\vdash \langle rs,$ *Undecided*$\rangle \Rightarrow$ *Decision FinalAllow*$\}$
using *FinalAllow-approximating-in-doubt-deny* **by** (*metis* (*lifting*, *full-types*) *Collect-mono*)

11.3 Approximating Closures

theorem *FinalAllowClosure*:
assumes *matcher-agree-on-exact-matches* γ β **and** *good-ruleset rs*
shows $\{p.$ (β, *in-doubt-deny*),$p\vdash \langle rs,$ *Undecided*$\rangle \Rightarrow_\alpha$ *Decision FinalAllow*$\} \subseteq$ $\{p.$ $\Gamma,\gamma,p\vdash \langle rs,$ *Undecided*$\rangle \Rightarrow$ *Decision FinalAllow*$\}$
and $\{p.$ $\Gamma,\gamma,p\vdash \langle rs,$ *Undecided*$\rangle \Rightarrow$ *Decision FinalAllow*$\} \subseteq \{p.$ (β, *in-doubt-allow*),$p\vdash$ $\langle rs,$ *Undecided*$\rangle \Rightarrow_\alpha$ *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 $\{p.$ (β, *in-doubt-allow*),$p\vdash \langle rs,$ *Undecided*$\rangle \Rightarrow_\alpha$ *Decision FinalDeny*$\} \subseteq$ $\{p.$ $\Gamma,\gamma,p\vdash \langle rs,$ *Undecided*$\rangle \Rightarrow$ *Decision FinalDeny*$\}$
and $\{p.$ $\Gamma,\gamma,p\vdash \langle rs,$ *Undecided*$\rangle \Rightarrow$ *Decision FinalDeny*$\} \subseteq \{p.$ (β, *in-doubt-deny*),$p\vdash$ $\langle rs,$ *Undecided*$\rangle \Rightarrow_\alpha$ *Decision FinalDeny*$\}$
apply (*metis FinalDenys-subseteq-in-doubt-allow assms*)
by (*metis FinalDeny-approximating-in-doubt-deny assms mem-Collect-eq subsetI*)

11.4 Exact Embedding

thm *matcher-agree-on-exact-matches-def*[*of* γ β]
lemma *LukassLemma*:
matcher-agree-on-exact-matches γ $\beta \Longrightarrow$

($\forall$ $r \in$ *set rs*. *ternary-ternary-eval* (*map-match-tac* β *p* (*get-match r*)) $\neq$ *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-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-bigstep-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 *Fixed-Action*
imports *Semantics-Ternary*
begin

12 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* (λ*m*. *Rule m Log*) *ms*) *Undecided* = *Undecided*
apply(*induction ms*, *simp-all*)
done
lemma *fixedaction-Empty*:*approximating-bigstep-fun* γ *p* (*map* (λ*m*. *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* ((λ*m*. *Rule m Log*) $\circ$ *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* (λm. *Rule m a*) (*m1*@*m2*)) *s* = *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) (*m2*@*m1*)) *s*
proof(*cases s*)
case *Decision* **thus** *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) (*m1* @ *m2*)) *s* = *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) (*m2* @ *m1*)) *s*
by(*simp add*: *Decision-approximating-bigstep-fun*)
next
case *Undecided*
have *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) *m1* @ *map* (λm. *Rule m a*) *m2*) *Undecided* = *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) *m2* @ *map* (λm. *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* (λm. *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* (λm. *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* ∈ *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* ∉ *set m1* **.**
show *?goal*
proof (*cases m* ∉ *set m2*)
case *True*
from *True* ‹*m* ∉ *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* ∈ *set m2* **by** *simp*

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

from *Cons.prems* ‹*m* ∉ *set m1*› **have** *set m1* = *set* (*filter* (λ*x*. *x*≠*m*) *m2*) **by** *auto*
from *Cons.IH*[*OF this*] **have** *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) *m1*) *Undecided* = *approximating-bigstep-fun* γ *p* (*map* (λ*m*. *Rule m a*) [*x*←*m2* . *x* ≠ *m*]) *Undecided* **.**
from *this* **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*)

(m#[x←$m2$. $x \neq m$])) *Undecided*
apply(*simp split*: *action.split*)
by *fast*
also have ... = *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) ([x←$m2$. $x = m$]@[x←$m2$. $x \neq m$])) *Undecided*
apply(*simp only*: *list.map*)
thm *approximating-bigstep-fun-prepend-replicate*[**where** *n*=*length* [x←$m2$. $x = m$]]
apply(*subst approximating-bigstep-fun-prepend-replicate*[**where** *n*=*length* [x←$m2$. $x = m$]])
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←$m2'$. $x = m$] @ *Rule m2'1 a* # *map* (λm. *Rule m a*) [x←$m2'$. $x \neq m$]) *Undecided* =
approximating-bigstep-fun γ *p* (*map* (λm. *Rule m a*) ([x←$m2'$. $x = m$] @ [*m2'1*] @ [x←$m2'$. $x \neq m$])) *Undecided* **by** *fastforce*
also have ... = *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) ([*m2'1*] @ [x←$m2'$. $x = m$] @ [x←$m2'$. $x \neq m$])) *Undecided*
using *fixedaction-reorder* **by** *fast*
finally have *XX*: *approximating-bigstep-fun* γ *p* (*map* (λm. *Rule m a*) [x←$m2'$. $x = m$] @ *Rule m2'1 a* # *map* (λm. *Rule m a*) [x←$m2'$. $x \neq m$]) *Undecided* =
approximating-bigstep-fun γ *p* (*Rule m2'1 a* # (*map* (λm. *Rule m a*) [x←$m2'$. $x = m$] @ *map* (λm. *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

12.1 *match-list*

Reducing the firewall semantics to shortcircuit 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 ⟷ (∃ m ∈ 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 (λm. 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: ¬ match-list γ ms a p ⟹ approximating-bigstep-fun γ
p (map (λm. Rule m a) ms) Undecided = Undecided
  apply(induction ms)
   apply(simp)
  apply(simp split: split-if-asm action.split)
  done

lemma match-list-semantics: match-list γ ms1 a p ⟷ match-list γ ms2 a p
⟹
  approximating-bigstep-fun γ p (map (λm. Rule m a) ms1) s = approximating-bigstep-fun
γ p (map (λm. Rule m a) ms2) s
  apply(case-tac s)
   prefer 2
   apply(simp add: Decision-approximating-bigstep-fun)
  apply(simp)
  apply(thin-tac s = ?un)
  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

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

lemma *empty-concat*: (*concat* (*map* ($\lambda x.$ []) *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* ($\lambda x.$ *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* ($\lambda x.$ *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*: ¬ *match-list* γ *m1'* *a p* $\Longrightarrow$ ¬ *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* ¬ *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* (λ*m*. *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*)

13 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** (*simp*) (*metis ex-in-conv matches-simp2 matches-simp22 set-empty*)
next
case *4* **thus** *?case* **using** *match-list-normalize-match* **by** (*metis match-list.simps*)
next
case *5* **thus** *?case* **using** *matches-not-idem* **by** *fastforce*
next
case *6* **thus** *?case* **by** (*metis bunch-of-lemmata-about-matches*(*3*) *matches-def matches-tuple*)
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* (*λm. 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*
    apply(*simp add*: *fixedaction-wf-ruleset*)
    apply(*unfold wf-ruleset-singleton*)
    apply(*simp add*: *matches-to-match-list-normalize*)
    done
  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*
    apply(*unfold wf-ruleset-singleton*)
    apply(*simp add*: *matches-to-match-list-normalize*)
    done
  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* (*λm. 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*
    apply(*simp add*: *fixedaction-wf-ruleset*)
    apply(*unfold wf-ruleset-singleton*)
    apply(*simp add*: *matches-to-match-list-normalize*)
    done
  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*
apply(*unfold wf-ruleset-singleton*)
apply(*simp add*: *matches-to-match-list-normalize*)
done
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

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

lemma *normalize-rules-singleton*: *normalize-rules* [*Rule m a*] = *map* (λ*m*. *Rule m a*) (*normalize-match m*) **by** *simp*

lemma *normalize-rules-fst*: (*normalize-rules* (*r* # *rs*)) = (*normalize-rules* [*r*]) @ (*normalize-rules rs*)
by(*cases r*) (*simp*)

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

lemma *wf-ruleset-normalize-rules*: *wf-ruleset* γ *p rs* ⟹ *wf-ruleset* γ *p* (*normalize-rules 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 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* [*r*]) **using** *wf-ruleset-normalize-match* **by**(*cases r*) *simp*

with *IH wf-ruleset-append* **have** *wf-ruleset* γ *p* (*normalize-rules* [*r*] @ *normalize-rules rs*) **by** *fast*
thus *?case* **by**(*subst normalize-rules-fst*)
qed

lemma *good-ruleset-normalize-rules*: *good-ruleset rs* $\Longrightarrow$ *good-ruleset* (*normalize-rules rs*)
proof(*induction rs*)
case *Nil* **thus** *?case* **by** (*simp add*: *good-ruleset-tail*)
next
case(*Cons r rs*)
from *Cons* **have** *IH*: *good-ruleset* (*normalize-rules 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* [*r*]) **by**(*cases r*) (*simp add*: *good-ruleset-normalize-match*)
with *IH good-ruleset-append* **have** *good-ruleset* (*normalize-rules* [*r*] @ *normalize-rules rs*) **by** *blast*
thus *?case* **by**(*subst normalize-rules-fst*)
qed

lemma *normalize-rules-correct*: *wf-ruleset* γ *p rs* $\Longrightarrow$ *approximating-bigstep-fun* γ *p* (*normalize-rules 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 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* [*r*])
by(*auto dest*: *wf-rulesetD simp*: *wf-ruleset-normalize-rules*)
with *IH Undecided* **have**
approximating-bigstep-fun γ *p* (*normalize-rules rs*) (*approximating-bigstep-fun* γ *p* (*normalize-rules* [*r*]) *Undecided*) = *approximating-bigstep-fun* γ *p* (*r* # *rs*) *Undecided*
apply(*case-tac 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* [*r*] @ *normalize-rules rs*)

$s = approximating\text{-}bigstep\text{-}fun\ \gamma\ p\ (r\ \#\ rs)\ s$
using *Undecided* ‹*wf-ruleset* γ *p* [*r*]› ‹*wf-ruleset* γ *p* (*normalize-rules* [*r*])›
by(*simp add*: *approximating-bigstep-fun-seq-wf*)
thus *?goal* **using** *normalize-rules-fst* **by** *metis*
qed
qed

fun *normalized-match* :: *'a match-expr* $\Rightarrow$ *bool* **where**
normalized-match MatchAny = *True* |
normalized-match (*Match* -) = *True* |
normalized-match (*MatchNot* (*Match* -)) = *True* |
normalized-match (*MatchAnd m1 m2*) = ((*normalized-match m1*) $\wedge$ (*normalized-match m2*)) |
normalized-match - = *False*

Essentially, *normalized-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-match-normalize-match*: $\forall$ *m'* $\in$ *set* (*normalize-match m*). *normalized-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*

end
theory *Iptables-Semantics*
imports *Semantics-Embeddings Fixed-Action*
begin

14 Normalizing Rulesets in the Boolean Big Step Semantics

corollary *normalize-rules-correct-BooleanSemantics*:
assumes *good-ruleset rs*
shows $\Gamma,\gamma,p\vdash \langle normalize\text{-}rules\ rs,\ s\rangle \Rightarrow t \longleftrightarrow \Gamma,\gamma,p\vdash \langle rs,\ s\rangle \Rightarrow t$
proof –
from *assms* **have** *assm'*: *good-ruleset* (*normalize-rules rs*) **by** (*metis good-ruleset-normalize-rules*)

from *normalize-rules-correct assms good-imp-wf-ruleset* **have**
$\forall\, \beta\ \alpha.$ *approximating-bigstep-fun* (β,α) *p* (*normalize-rules rs*) *s* = *approximating-bigstep-fun* (β,α) *p rs s* **by** *fast*
hence
$\forall\, \alpha.$ *approximating-bigstep-fun* $(\beta_{magic}\ \gamma,\alpha)$ *p* (*normalize-rules rs*) *s* = *approximating-bigstep-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 *Negation-Type*
imports *Main*
begin

15 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* (λ*x*::*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*) $\subseteq$ *set x*
apply(*induction x rule*: *getPos.induct*)
apply(*simp-all*)
apply *blast+*
done
lemma *set-Neg-getNeg-subset*: *Neg* ‘ *set* (*getNeg x*) $\subseteq$ *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*

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

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

16 Negation Type Matching

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

fun *alist-and* :: $'a$ *negation-type list* $\Rightarrow$ $'a$ *match-expr* **where**
alist-and [] = *MatchAny* |
alist-and ((*Pos e*)#*es*) = *MatchAnd* (*Match e*) (*alist-and es*) |
alist-and ((*Neg e*)#*es*) = *MatchAnd* (*MatchNot* (*Match e*)) (*alist-and es*)

fun *negation-type-to-match-expr* :: $'a$ *negation-type* $\Rightarrow$ $'a$ *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*)

lemma *alist-and-append*: *matches* γ (*alist-and* (*l1* @ *l2*)) *a p* $\longleftrightarrow$ *matches* γ (*MatchAnd* (*alist-and l1*) (*alist-and l2*)) *a p*
apply(*induction l1*)
apply(*simp-all add*: *bunch-of-lemmata-about-matches*)
apply(*rename-tac l l1*)
apply(*case-tac l*)
apply(*simp-all add*: *bunch-of-lemmata-about-matches*)
done

fun *to-negation-type-nnf* :: $'a$ *match-expr* $\Rightarrow$ $'a$ *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*)

lemma *normalized-match m* $\Longrightarrow$ *matches* γ (*alist-and* (*to-negation-type-nnf m*)) *a p* = *matches* γ *m a p*
apply(*induction m rule*: *to-negation-type-nnf.induct*)
apply(*simp-all add*: *bunch-of-lemmata-about-matches alist-and-append*)
done

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(*simp-all*)
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*)
apply(*induction* γ *a p ms rule*: *nt-match-list.induct*)
apply(*simp-all*)
by *fastforce*

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*)
by (*metis* (*poly-guards-query*) *nt-match-list-matches nt-match-list-simp*)

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* $\Rightarrow$ *bool*) $\Rightarrow$ '*a match-expr* $\Rightarrow$ *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* $\vee$ *has-disc disc m2*)

The following function takes a tuple of functions (('*a* $\Rightarrow$ *bool*) $\times$ ('*a* $\Rightarrow$ '*b*)) and a *'a match-expr*. The passed function tuple must be the discriminator and selector of the datatype package. *primitive-extractor* filters the *'a match-expr* and returns a tuple. The first element of the returned tuple is the filtered primitive matches, the second element is the remaining match expression.

It requires a *normalized-match*.

fun *primitive-extractor* :: $(('a \Rightarrow bool) \times ('a \Rightarrow 'b)) \Rightarrow 'a$ *match-expr* $\Rightarrow$ *('b negation-type list* $\times$ *'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'*))

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-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-match ms*
and $\neg$ *has-disc disc ms*
and $\forall$ *disc2*. $\neg$ *has-disc disc2 m* $\longrightarrow$ $\neg$ *has-disc disc2 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*
apply(*induction* (*disc*, *sel*) *m arbitrary*: *as ms rule*: *primitive-extractor.induct*)
apply(*simp-all add*: *bunch-of-lemmata-about-matches wf-disc-sel.simps split*: *split-if-asm*)
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

from *assms*(*1*) *assm3'* **show** *normalized-match ms*
apply(*induction* (*disc*, *sel*) *m arbitrary*: *as ms rule*: *primitive-extractor.induct*)
apply(*simp*)

apply(*simp*)
apply(*simp split*: *split-if-asm*)
apply(*simp split*: *split-if-asm*)
apply(*clarify*)
apply(*simp split*: *split-split-asm*)
apply(*simp*)
apply(*simp*)
apply(*simp*)
done

from *assms*(*1*) *assm3'* **show** $\neg$ *has-disc disc ms*
apply(*induction* (*disc*, *sel*) *m* *arbitrary*: *as ms rule*: *primitive-extractor.induct*)
by(*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*
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(*clarify*)
apply(*simp split*: *split-split-asm*)
apply(*simp-all*)
done
qed

lemma *primitive-extractor-matchesE*: *wf-disc-sel* (*disc*,*sel*) *C* $\Longrightarrow$ *normalized-match m* $\Longrightarrow$ *primitive-extractor* (*disc*, *sel*) *m* = (*as*, *ms*)
$\Longrightarrow$
(*normalized-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-match m* $\Longrightarrow$ *primitive-extractor* (*disc*, *sel*) *m* = (*as*, *ms*)
$\Longrightarrow$
(*normalized-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-match ?m*; *primitive-extractor*

(*?disc*, *?sel*) *?m* = (*?as*, *?ms*); ⟦*normalized-match ?ms*; ¬ *has-disc ?disc ?ms*; ∀ *disc2*. ¬ *has-disc disc2 ?m* ⟶ ¬ *has-disc disc2 ?ms*⟧ ⟹ *?matches-other* = *matches ?γ ?ms ?a ?p*⟧ ⟹ (*matches ?γ* (*alist-and* (*NegPos-map ?C ?as*)) *?a ?p* ∧ *?matches-other*) = *matches ?γ ?m ?a ?p* and ⟦*wf-disc-sel* (*?disc*, *?sel*) *?C*; *normalized-match ?m*; *primitive-extractor* (*?disc*, *?sel*) *?m* = (*?as*, *?ms*); ⟦*normalized-match ?ms*; ¬ *has-disc ?disc ?ms*; ∀ *disc2*. ¬ *has-disc disc2 ?m* ⟶ ¬ *has-disc disc2 ?ms*⟧ ⟹ *matches ?γ ?ms ?a ?p*⟧ ⟹ *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*].

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

17 Util: listprod

definition *listprod* :: *nat list* ⇒ *nat* **where** *listprod as* ≡ *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* ⟹ *listprod as* = ∏ (*set as*)
by(*induction as*) *simp-all*

18 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-new *'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 `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* -) = ($\lambda m.$ ¬ *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* ≡ ⋃ *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* = (⋃ *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* = (⋃ *ms* ∈ *set* (*packet-set-repr ps*).
(⋂ (*m*, *a*) ∈ *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* ∈ *packet-set-to-set* γ (*to-packet-set a m*) ⟷ *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 ≡ *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 ≡ *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 ∀ *p*. *matches* γ *m a1 p* ⟷ *matches* γ *m a2 p* ⟹ *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

18.0.1 Basic Set Operations

∩

fun *packet-set-intersect* :: *'a packet-set* ⇒ *'a packet-set* ⇒ *'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* ∩ *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* ($\lambda a.$ [*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* ($\lambda a.$ [*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*) = *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

18.0.2 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*) = $\{p \in$ *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*

18.0.3 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-type* × *action negation-type*) *list*) *list* ⇒ ((*'a negation-type* × *action negation-type*) *list*) *list* **where**
packet-set-opt2-internal [] = [] |

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

packet-set-opt2-internal (*as*#*ps*) = *as*# (*if length as* ≤ *5 then packet-set-opt2-internal* ((*filter* (λ*ass*. ¬ *set as* ⊆ *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* ⇒ *'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* ⇒ *'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-type* × *action negation-type*) *list*) ⇒ *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* $\Rightarrow$ *'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*

18.1 Conjunction Normal Form Packet Set

datatype-new *'a packet-set-cnf* = *PacketSetCNF* (*packet-set-repr-cnf*: ((*'a negation-type* $\times$ *action negation-type*) *list*) *list*)

lemma $\neg$ (($a \wedge b$) $\vee$ ($c \wedge d$)) $\longleftrightarrow$ ($\neg a \vee \neg b$) $\wedge$ ($\neg c \vee \neg d$) **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* $\Rightarrow$ *'a packet-set-cnf* $\Rightarrow$ *'packet set* **where**
packet-set-cnf-to-set γ *ps* $\equiv$ ($\bigcap$ *ms* $\in$ *set* (*packet-set-repr-cnf ps*).
($\bigcup$ (*m*, *a*) $\in$ *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* $\Rightarrow$ *'a packet-set-cnf* **where**
packet-set-not-to-cnf (*PacketSet ps*) = *PacketSetCNF* (*map* (λa. *map invertt a*) *ps*)
declare *packet-set-not-to-cnf.simps*[*simp del*]

lemma *helper*: (*case invertt x of* (*m*, *a*) $\Rightarrow$ {*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*) $\Rightarrow$ {*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* $\Rightarrow$ *'a packet-set* **where**
packet-set-cnf-not-to-dnf (*PacketSetCNF ps*) = *PacketSet* (*map* (λa. *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* $\Rightarrow$ *'a packet-set-cnf* $\Rightarrow$ *'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*. ¬ *set as* ⊆ *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* ⇒ ′*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* ⇒ ′*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 *Optimizing*
imports *Semantics-Ternary Packet-Set-Impl*
begin

19 Optimizing

19.1 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*$\}$))

19.1.1 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* ⟹ *?y*)
apply(*thin-tac a* = *?x* ⟹ *?y*)
by (*metis bunch-of-lemmata-about-matches*(*3*) *surj-pair*)

end
theory *IPSpace-Operations*
imports *Negation-Type ../Bitmagic/Numberwang-Ln ../Primitive-Matchers/IPSpace-Syntax ../Bitmagic/IPv4Addr*
begin

definition *intersect-netmask-empty* :: *nat* × *nat* × *nat* × *nat* ⇒ *nat* ⇒ *nat* × *nat* × *nat* × *nat* ⇒ *nat* ⇒ *bool* **where**
intersect-netmask-empty base1 m1 base2 m2 ≡
ipv4range-set-from-bitmask (*ipv4addr-of-dotteddecimal base1*) *m1* ∩ *ipv4range-set-from-bitmask* (*ipv4addr-of-dotteddecimal base2*) *m2* = {}

fun *ipv4range-set-from-bitmask-to-executable-ipv4range* :: *ipt-ipv4range* ⇒ *32 bitrange* **where**
ipv4range-set-from-bitmask-to-executable-ipv4range (*Ip4AddrNetmask pre len*) =

ipv4range-range (((*ipv4addr-of-dotteddecimal pre*) *AND* ((*mask len*) << (*32* − *len*)))) ((*ipv4addr-of-dotteddecimal pre*) *OR* (*mask* (*32* − *len*))) |
ipv4range-set-from-bitmask-to-executable-ipv4range (*Ip4Addr ip*) = *ipv4range-single* (*ipv4addr-of-dotteddecimal ip*)

lemma *ipv4range-set-from-bitmask-to-executable-ipv4range-simps*:
ipv4range-to-set (*ipv4range-set-from-bitmask-to-executable-ipv4range* (*Ip4AddrNetmask base m*)) =
ipv4range-set-from-bitmask (*ipv4addr-of-dotteddecimal base*) *m*
ipv4range-to-set (*ipv4range-set-from-bitmask-to-executable-ipv4range* (*Ip4Addr ip*)) = {*ipv4addr-of-dotteddecimal ip*}
by(*simp-all add*: *ipv4range-set-from-bitmask-alt ipv4range-range-set-eq ipv4range-single-set-eq*)

declare *ipv4range-set-from-bitmask-to-executable-ipv4range.simps*[*simp del*]

definition *intersect-netmask-empty-executable* ≡ (λ *base1 m1 base2 m2*. *ipv4range-empty* (
ipv4range-intersection
(*ipv4range-set-from-bitmask-to-executable-ipv4range* (*Ip4AddrNetmask base1 m1*))
(*ipv4range-set-from-bitmask-to-executable-ipv4range* (*Ip4AddrNetmask base2 m2*))))

lemma [*code*]: *intersect-netmask-empty* = *intersect-netmask-empty-executable*
apply (*rule ext*)+
unfolding *intersect-netmask-empty-def intersect-netmask-empty-executable-def*
apply(*simp add*: *ipv4range-set-from-bitmask-to-executable-ipv4range-simps*)
done

export-code *intersect-netmask-empty* **in** *SML*

definition *subset-netmask* :: *nat* × *nat* × *nat* × *nat* ⇒ *nat* ⇒ *nat* × *nat* × *nat* × *nat* ⇒ *nat* ⇒ *bool* **where**
subset-netmask base1 m1 base2 m2 ≡
ipv4range-set-from-bitmask (*ipv4addr-of-dotteddecimal base1*) *m1* ⊆ *ipv4range-set-from-bitmask* (*ipv4addr-of-dotteddecimal base2*) *m2*

definition *subset-netmask-executable* :: *nat* × *nat* × *nat* × *nat* ⇒ *nat* ⇒ *nat* × *nat* × *nat* × *nat* ⇒ *nat* ⇒ *bool* **where**
subset-netmask-executable ≡ (λ *base1 m1 base2 m2*. *ipv4range-subset*
(*ipv4range-set-from-bitmask-to-executable-ipv4range* (*Ip4AddrNetmask base1 m1*))
(*ipv4range-set-from-bitmask-to-executable-ipv4range* (*Ip4AddrNetmask base2 m2*)))

lemma [*code*]: *subset-netmask* = *subset-netmask-executable*

apply(*simp only*: *fun-eq-iff*, *intro allI*)
unfolding *subset-netmask-def subset-netmask-executable-def*
apply(*simp add*: *ipv4range-set-from-bitmask-to-executable-ipv4range-simps*)
done

fun *intersect-ips* :: *ipt-ipv4range* ⇒ *ipt-ipv4range* ⇒ *ipt-ipv4range option* **where**
intersect-ips (*Ip4Addr ip*) (*Ip4AddrNetmask base m*) =
(*if* (*ipv4addr-of-dotteddecimal ip*) ∈ (*ipv4range-set-from-bitmask* (*ipv4addr-of-dotteddecimal base*) *m*)
then
Some (*Ip4Addr ip*)
else
None) |
intersect-ips (*Ip4AddrNetmask base m*) (*Ip4Addr ip*) =
(*if* (*ipv4addr-of-dotteddecimal ip*) ∈ (*ipv4range-set-from-bitmask* (*ipv4addr-of-dotteddecimal base*) *m*)
then
Some (*Ip4Addr ip*)
else
None) |
intersect-ips (*Ip4Addr ip1*) (*Ip4Addr ip2*) =
(*if ipv4addr-of-dotteddecimal ip2* = *ipv4addr-of-dotteddecimal ip1* (∗*there might be overflows if someone uses values* > *256*∗)
then
Some (*Ip4Addr ip1*)
else
None) |
intersect-ips (*Ip4AddrNetmask base1 m1*) (*Ip4AddrNetmask base2 m2*) =
(*if* (∗*ipv4range-set-from-bitmask* (*ipv4addr-of-dotteddecimal base1*) *m1* ∩ *ipv4range-set-from-bitmask* (*ipv4addr-of-dotteddecimal base2*) *m2* = {}∗)
intersect-netmask-empty base1 m1 base2 m2
then
None
else if (∗*m1* ≥ *m2*∗) (∗*maybe use execuatble subset check to make proofs easier?*∗)
subset-netmask base1 m1 base2 m2
then
Some (*Ip4AddrNetmask base1 m1*) (∗*andersrum?*∗)
else if subset-netmask base2 m2 base1 m1 then
Some (*Ip4AddrNetmask base2 m2*)
else
None (∗*cannot happen*, *one must be subset of each other*∗))

export-code *intersect-ips* **in** *SML*

lemma $\neg$ *ipv4range-set-from-bitmask b2 m2* $\subseteq$ *ipv4range-set-from-bitmask b1 m1* $\longrightarrow$
$\neg$ *ipv4range-set-from-bitmask b1 m1* $\subseteq$ *ipv4range-set-from-bitmask b2 m2* $\longrightarrow$
ipv4range-set-from-bitmask b1 m1 $\cap$ *ipv4range-set-from-bitmask b2 m2* $= \{\}$
using *ipv4range-bitmask-intersect* **by** *auto*

lemma *intersect-ips-None*: *intersect-ips ip1 ip2* $=$ *None* $\longleftrightarrow$ (*ipv4s-to-set ip1*) $\cap$ (*ipv4s-to-set ip2*) $= \{\}$
apply(*induction ip1 ip2 rule*: *intersect-ips.induct*)
apply(*simp-all add*: *intersect-netmask-empty-def subset-netmask-def ipv4range-bitmask-intersect*)
done

lemma *intersect-ips-Some*: *intersect-ips ip1 ip2* $=$ *Some X* $\Longrightarrow$ (*ipv4s-to-set ip1*) $\cap$ (*ipv4s-to-set ip2*) $=$ *ipv4s-to-set X*
apply(*induction ip1 ip2 rule*: *intersect-ips.induct*)
apply(*case-tac* [!] *X*)
apply(*auto simp add*: *ipv4range-set-from-bitmask-to-executable-ipv4range-simps intersect-netmask-empty-def subset-netmask-def split*: *split-if-asm*)
done

The other direction does not directly hold. Someone might enter some invalid ips.

lemma *intersect-ips-Some2*: (*ipv4s-to-set ip1*) $\cap$ (*ipv4s-to-set ip2*) $=$ *ipv4s-to-set X* $\Longrightarrow \exists$ *Y*. *intersect-ips ip1 ip2* $=$ *Some Y* $\wedge$ *ipv4s-to-set X* $=$ *ipv4s-to-set Y*
proof $-$
assume *a*: (*ipv4s-to-set ip1*) $\cap$ (*ipv4s-to-set ip2*) $=$ *ipv4s-to-set X*
hence (*ipv4s-to-set ip1*) $\cap$ (*ipv4s-to-set ip2*) $\neq \{\}$ **by**(*simp add*: *ipv4s-to-set-nonempty*)
with *a* **have** *ipv4s-to-set X* $\neq \{\}$ **by**(*simp add*: *intersect-ips-None*)
with *a* **have** *intersect-ips ip1 ip2* $\neq$ *None* **by**(*simp add*: *intersect-ips-None*)
from *this* **obtain** *Y* **where** *intersect-ips ip1 ip2* $=$ *Some Y* **by** *blast*
with *a intersect-ips-Some* **have** *intersect-ips ip1 ip2* $=$ *Some Y* $\wedge$ *ipv4s-to-set X* $=$ *ipv4s-to-set Y* **by** *simp*
thus *?thesis* **by** *blast*
qed

fun *compress-pos-ips* :: *ipt-ipv4range list* $\Rightarrow$ *ipt-ipv4range option* **where**
compress-pos-ips [] $=$ *Some* (*Ip4AddrNetmask* (*0,0,0,0*) *0*) |
compress-pos-ips [*ip*] $=$ *Some ip* |
compress-pos-ips (*a#b#cs*) $=$ (
case intersect-ips a b of None $\Rightarrow$ *None*
| *Some x* $\Rightarrow$ *compress-pos-ips* (*x#cs*)
)

lemma *compress-pos-ips-None*: *compress-pos-ips ips* = *None* $\longleftrightarrow \bigcap$ (*ipv4s-to-set* ‘ *set ips*) = {}
  apply(*induction ips rule*: *compress-pos-ips.induct*)
    apply(*simp*)
   apply(*simp add*: *ipv4s-to-set-nonempty*)
  apply(*simp*)
  apply(*simp split*: *option.split*)
  apply(*simp add*: *intersect-ips-None*)
  using *intersect-ips-Some* **by** *blast*

lemma *compress-pos-ips-Some*: *compress-pos-ips ips* = *Some X* $\Longrightarrow \bigcap$ (*ipv4s-to-set* ‘ *set ips*) = *ipv4s-to-set X*
  apply(*induction ips rule*: *compress-pos-ips.induct*)
    apply(*simp*)
    apply(*auto simp add*: *ipv4range-set-from-bitmask-0*)[*1*]
   apply(*simp*)
  apply(*simp*)
  apply(*simp split*: *option.split-asm*)
by (*metis Int-assoc intersect-ips-Some*)

fun *collect-to-range* :: *ipt-ipv4range list* ⇒ *32 bitrange* **where**
 collect-to-range [] = *Empty-Bitrange* |
 collect-to-range (*r*#*rs*) = *RangeUnion* (*ipv4range-set-from-bitmask-to-executable-ipv4range r*) (*collect-to-range rs*)

fun *compress-ips* :: *ipt-ipv4range negation-type list* ⇒ *ipt-ipv4range negation-type list option* **where**
  compress-ips l = (*if* (*getPos l*) = [] *then Some l* (∗*fix not to introduce* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*), *only return the negative list*∗)
   else
  (*case compress-pos-ips* (*getPos l*)
    of None ⇒ *None*
    | *Some ip* ⇒
    if ipv4range-empty (*ipv4range-setminus* (*ipv4range-set-from-bitmask-to-executable-ipv4range ip*) (*collect-to-range* (*getNeg l*)))
      (∗ $\bigcap$ *pos* − $\bigcup$ *neg* = {}∗)
      then
        None
      else Some (*Pos ip* # *map Neg* (*getNeg l*))
      ))

export-code *compress-ips* **in** *SML*

lemma *ipv4range-set-from-bitmask-to-executable-ipv4range*:
ipv4range-to-set (*ipv4range-set-from-bitmask-to-executable-ipv4range a*) = *ipv4s-to-set a*
apply(*case-tac a*)
apply(*simp-all add*:*ipv4range-set-from-bitmask-to-executable-ipv4range-simps*)
done

lemma *ipv4range-to-set-collect-to-range*: *ipv4range-to-set* (*collect-to-range ips*) = ($\bigcup$ *x*∈*set ips*. *ipv4s-to-set x*)
apply(*induction ips*)
apply(*simp add*: *ipv4range-to-set-def*)
apply(*simp add*: *ipv4range-set-from-bitmask-to-executable-ipv4range ipv4range-to-set-def*)
by (*metis ipv4range-set-from-bitmask-to-executable-ipv4range ipv4range-to-set-def*)

lemma *compress-ips-None*: *getPos ips* ≠ [] ⟹ *compress-ips ips* = *None* ⟷ ($\bigcap$ (*ipv4s-to-set* ' *set* (*getPos ips*))) − ($\bigcup$ (*ipv4s-to-set* ' *set* (*getNeg ips*))) = {}
apply(*simp split*: *split-if*)
apply(*simp*)

apply(*simp split*: *option.split*)
apply(*intro conjI impI allI*)
apply(*simp add*: *compress-pos-ips-None*)
apply(*rename-tac a*)
apply(*frule compress-pos-ips-Some*)
apply(*case-tac a*)
apply(*simp add*: *ipv4range-set-from-bitmask-to-executable-ipv4range-simps*)
apply(*simp add*: *ipv4range-to-set-collect-to-range ipv4range-set-from-bitmask-to-executable-ipv4range-simps*)
apply(*simp add*: *ipv4range-to-set-collect-to-range ipv4range-set-from-bitmask-to-executable-ipv4range-simps*)
apply(*rename-tac a*)
apply(*frule compress-pos-ips-Some*)
apply(*case-tac a*)
apply(*simp add*: *ipv4range-to-set-collect-to-range ipv4range-set-from-bitmask-to-executable-ipv4range-simps*)
apply(*simp add*: *ipv4range-to-set-collect-to-range ipv4range-set-from-bitmask-to-executable-ipv4range-simps*)
done

lemma *compress-ips-emptyPos*: *getPos ips* = [] ⟹ *compress-ips ips* = *Some ips* ∧ *ips* = *map Neg* (*getNeg ips*)
apply(*simp only*: *compress-ips.simps split*: *split-if*)
apply(*intro conjI impI*)
apply(*simp-all*)
apply(*induction ips*)
apply(*simp-all*)
apply(*case-tac a*)
apply(*simp-all*)

done

end
theory *Format-Ln*
imports *Negation-Type-Matching ../Bitmagic/Numberwang-Ln ../Primitive-Matchers/IPSpace-Syntax ../Bitmagic/IPv4Addr*
begin

20 iptables LN formatting

Produce output as produced by the command: iptables -L -n

Example

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
STATEFUL   all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0            icmptype 3
...
```

datatype *match-Ln-uncompressed* = *UncompressedFormattedMatch*
ipt-ipv4range negation-type list
ipt-ipv4range negation-type list
ipt-protocol negation-type list
string negation-type list

fun *UncompressedFormattedMatch-to-match-expr* :: *match-Ln-uncompressed* ⇒ *iptrule-match match-expr* **where**
UncompressedFormattedMatch-to-match-expr (*UncompressedFormattedMatch src dst proto extra*) =
MatchAnd (*alist-and* (*NegPos-map Src src*)) (*MatchAnd* (*alist-and* (*NegPos-map Dst dst*)) (*MatchAnd* (*alist-and* (*NegPos-map Prot proto*)) (*alist-and* (*NegPos-map Extra extra*))))

append

fun *match-Ln-uncompressed-append* :: *match-Ln-uncompressed* ⇒ *match-Ln-uncompressed* ⇒ *match-Ln-uncompressed* **where**
match-Ln-uncompressed-append (*UncompressedFormattedMatch src1 dst1 proto1 extra1*) (*UncompressedFormattedMatch src2 dst2 proto2 extra2*) =
UncompressedFormattedMatch (*src1*@*src2*) (*dst1*@*dst2*) (*proto1*@*proto2*) (*extra1*@*extra2*)

lemma *matches-match-Ln-uncompressed-append*: *matches* γ (*UncompressedFormattedMatch-to-match-expr* (*match-Ln-uncompressed-append fmt1 fmt2*)) *a p* ⟷

matches γ (*MatchAnd* (*UncompressedFormattedMatch-to-match-expr fmt1*)
(*UncompressedFormattedMatch-to-match-expr fmt2*)) *a p*
apply(*case-tac fmt1*)
apply(*case-tac fmt2*)
apply(*clarify*)
apply(*simp*)
apply(*simp add*: *alist-and-append NegPos-map-append bunch-of-lemmata-about-matches*)
by *fastforce*

assumes: *normalized-match*

fun *iptrule-match-collect* :: *iptrule-match match-expr* $\Rightarrow$ *match-Ln-uncompressed* $\Rightarrow$ *match-Ln-uncompressed* **where**
iptrule-match-collect MatchAny accu = *accu* |
iptrule-match-collect (*Match* (*Src ip*)) (*UncompressedFormattedMatch src dst proto extra*) = *UncompressedFormattedMatch* ((*Pos ip*)#*src*) *dst proto extra* |
iptrule-match-collect (*Match* (*Dst ip*)) (*UncompressedFormattedMatch src dst proto extra*) = *UncompressedFormattedMatch src* ((*Pos ip*)#*dst*) *proto extra* |
iptrule-match-collect (*Match* (*Prot p*)) (*UncompressedFormattedMatch src dst proto extra*) = *UncompressedFormattedMatch src dst* ((*Pos p*)#*proto*) *extra* |
iptrule-match-collect (*Match* (*Extra e*)) (*UncompressedFormattedMatch src dst proto extra*) = *UncompressedFormattedMatch src dst proto* ((*Pos e*)#*extra*) |
iptrule-match-collect (*MatchNot* (*Match* (*Src ip*))) (*UncompressedFormattedMatch src dst proto extra*) = *UncompressedFormattedMatch* ((*Neg ip*)#*src*) *dst proto extra* |
iptrule-match-collect (*MatchNot* (*Match* (*Dst ip*))) (*UncompressedFormattedMatch src dst proto extra*) = *UncompressedFormattedMatch src* ((*Neg ip*)#*dst*) *proto extra* |
iptrule-match-collect (*MatchNot* (*Match* (*Prot p*))) (*UncompressedFormattedMatch src dst proto extra*) = *UncompressedFormattedMatch src dst* ((*Neg p*)#*proto*) *extra* |
iptrule-match-collect (*MatchNot* (*Match* (*Extra e*))) (*UncompressedFormattedMatch src dst proto extra*) = *UncompressedFormattedMatch src dst proto* ((*Neg e*)#*extra*) |
iptrule-match-collect (*MatchAnd m1 m2*) *fmt* =
match-Ln-uncompressed-append (*iptrule-match-collect m1 fmt*)
(*match-Ln-uncompressed-append* (*iptrule-match-collect m2 fmt*) *fmt*)

We can express *iptrule-match-collect* with *primitive-extractor*. Latter is more elegant. We keep the definition of *iptrule-match-collect* to show explicitly what we are doing here.

lemma *iptrule-match-collect-by-primitive-extractor*: *normalized-match m* $\Longrightarrow$ *iptrule-match-collect m* (*UncompressedFormattedMatch* [] [] [] []) = (
let (*srcs*, *m'*) = *primitive-extractor* (*is-Src*, *src-range*) *m*;
(*dsts*, *m''*) = *primitive-extractor* (*is-Dst*, *dst-range*) *m'*;
(*protos*, *m'''*) = *primitive-extractor* (*is-Prot*, *prot-sel*) *m''*;
(*extras*, -) = *primitive-extractor* (*is-Extra*, *extra-sel*) *m'''*
in UncompressedFormattedMatch srcs dsts protos extras
)
apply(*induction m UncompressedFormattedMatch* [] [] [] [] *rule*: *iptrule-match-collect.induct*)

apply(*simp-all*)
apply(*simp add*: *split*: *split-split-asm split-split*)
done

Example

lemma *iptrule-match-collect* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*, *0*, *0*, *0*) *8*))) (*Match* (*Prot ProtTCP*))) (*UncompressedFormattedMatch* [] [] [] []) =
UncompressedFormattedMatch [*Pos* (*Ip4AddrNetmask* (*0*, *0*, *0*, *0*) *8*)] [] [*Pos* *ProtTCP*] [] **by** *eval*

The empty *UncompressedFormattedMatch* always match

lemma *matches* γ (*UncompressedFormattedMatch-to-match-expr* (*UncompressedFormattedMatch* [] [] [] [])) *a p*
by(*simp add*: *bunch-of-lemmata-about-matches*)

lemma *UncompressedFormattedMatch-to-match-expr-correct*: **assumes** *normalized-match m* **and** *matches* γ (*UncompressedFormattedMatch-to-match-expr accu*) *a p* **shows**
matches γ (*UncompressedFormattedMatch-to-match-expr* (*iptrule-match-collect m accu*)) *a p* ⟷ *matches* γ *m a p*
using *assms* **apply** (*induction m accu arbitrary*: *rule*: *iptrule-match-collect.induct*)
apply(*case-tac* [!] γ)
apply (*simp add*: *eval-ternary-simps ip-in-ipv4range-set-from-bitmask-UNIV bunch-of-lemmata-about-matche*
apply (*simp add*: *eval-ternary-simps ip-in-ipv4range-set-from-bitmask-UNIV bunch-of-lemmata-about-matche*
apply (*simp add*: *eval-ternary-simps ip-in-ipv4range-set-from-bitmask-UNIV bunch-of-lemmata-about-matche*
apply (*simp add*: *eval-ternary-simps ip-in-ipv4range-set-from-bitmask-UNIV bunch-of-lemmata-about-matche*
apply (*simp add*: *eval-ternary-simps ip-in-ipv4range-set-from-bitmask-UNIV bunch-of-lemmata-about-matche*
apply (*simp add*: *eval-ternary-simps ip-in-ipv4range-set-from-bitmask-UNIV bunch-of-lemmata-about-matche*
apply (*simp add*: *eval-ternary-simps ip-in-ipv4range-set-from-bitmask-UNIV bunch-of-lemmata-about-matche*
apply (*simp add*: *eval-ternary-simps ip-in-ipv4range-set-from-bitmask-UNIV bunch-of-lemmata-about-matche*
apply (*simp add*: *eval-ternary-simps ip-in-ipv4range-set-from-bitmask-UNIV bunch-of-lemmata-about-matche*
apply (*simp add*: *eval-ternary-simps ip-in-ipv4range-set-from-bitmask-UNIV bunch-of-lemmata-about-matche*
apply(*simp add*: *matches-match-Ln-uncompressed-append bunch-of-lemmata-about-matches*)

apply(*simp-all*) — ¬ *normalized-match*
done

definition *format-Ln-match* :: *iptrule-match match-expr* ⇒ *match-Ln-uncompressed*
where
format-Ln-match m ≡ *iptrule-match-collect m* (*UncompressedFormattedMatch* [] [] [] [])

corollary *format-Ln-match-correct*: *normalized-match m* ⟹ *matches* γ (*UncompressedFormattedMatch-to-m*
(*format-Ln-match m*)) *a p* ⟷ *matches* γ *m a p*
unfolding *format-Ln-match-def*
apply(*rule UncompressedFormattedMatch-to-match-expr-correct*)
apply(*assumption*)
by(*simp add*: *bunch-of-lemmata-about-matches*)

We can also show the previous corollary by the correctness of *primitive-extractor*

corollary *normalized-match m* $\Longrightarrow$ *matches* γ (*UncompressedFormattedMatch-to-match-expr* (*format-Ln-match m*)) *a p* $\longleftrightarrow$ *matches* γ *m a p*
proof −
{**fix** *yc*
have *normalized-match yc* $\Longrightarrow$ $\neg$ *has-disc is-Dst yc* $\Longrightarrow$ $\neg$ *has-disc is-Prot yc* $\Longrightarrow$ $\neg$ *has-disc is-Extra yc* $\Longrightarrow$ $\neg$ *has-disc is-Src yc* $\Longrightarrow$ *matches* γ *yc a p*
apply(*induction yc*)
apply(*simp-all add:bunch-of-lemmata-about-matches*)
apply(*case-tac aa*)
apply(*simp-all*)
apply(*case-tac yc*)
apply(*simp-all*)
apply(*case-tac aa*)
apply(*simp-all*)
done
} **note** *yc-exhaust=this*
assume *normalized*: *normalized-match m*
{**fix** *asrc msrc adst mdst aprot mprot aextra mextra*
from *normalized* **have**
primitive-extractor (*is-Extra*, *extra-sel*) *mprot* = (*aextra*, *mextra*) $\Longrightarrow$
primitive-extractor (*is-Prot*, *prot-sel*) *mdst* = (*aprot*, *mprot*) $\Longrightarrow$
primitive-extractor (*is-Dst*, *dst-range*) *msrc* = (*adst*, *mdst*) $\Longrightarrow$
primitive-extractor (*is-Src*, *src-range*) *m* = (*asrc*, *msrc*) $\Longrightarrow$
matches γ (*alist-and* (*NegPos-map Src asrc*)) *a p* $\wedge$
matches γ (*alist-and* (*NegPos-map Dst adst*)) *a p* $\wedge$
matches γ (*alist-and* (*NegPos-map Prot aprot*)) *a p* $\wedge$
matches γ (*alist-and* (*NegPos-map Extra aextra*)) *a p* $\longleftrightarrow$ *matches* γ *m a p*
apply −
apply(*erule*(*1*) *primitive-extractor-matchesE*[*OF wf-disc-sel-iptrule-match*(*1*)])
apply(*erule*(*1*) *primitive-extractor-matchesE*[*OF wf-disc-sel-iptrule-match*(*2*)])
apply(*erule*(*1*) *primitive-extractor-matchesE*[*OF wf-disc-sel-iptrule-match*(*3*)])
apply(*erule*(*1*) *primitive-extractor-matches-lastE*[*OF wf-disc-sel-iptrule-match*(*4*)])
using *yc-exhaust* **by** *metis*
}
thus *?thesis*
unfolding *format-Ln-match-def*
unfolding *iptrule-match-collect-by-primitive-extractor*[*OF normalized*]
by(*simp split*: *split-split add*: *bunch-of-lemmata-about-matches*(*1*))
qed

lemma *format-Ln-match-correct'*: $\forall$ *m'* $\in$ *set ms*. *normalized-match m'* $\Longrightarrow$
approximating-bigstep-fun γ *p* (*map* (λ*m*. *Rule m a*) (*map* (λ*m'*. *UncompressedFormattedMatch-to-match-exp* (*format-Ln-match m'*)) *ms*)) *s* =
approximating-bigstep-fun γ *p* (*map* (λ*m*. *Rule m a*) *ms*) *s*
apply(*rule match-list-semantics*)
apply(*induction ms*)

apply(*simp*)
apply(*simp*)
by (*metis format-Ln-match-correct*)

definition *format-Ln-rules-uncompressed* :: *iptrule-match rule list* ⇒ (*match-Ln-uncompressed* × *action*) *list* **where**
format-Ln-rules-uncompressed rs = [((*format-Ln-match* (*get-match r*)), (*get-action r*)). *r* ← (*normalize-rules rs*)]

definition *Ln-rules-to-rule* :: (*match-Ln-uncompressed* × *action*) *list* ⇒ *iptrule-match rule list* **where**
Ln-rules-to-rule rs = [*case r of* (*m*,*a*) ⇒ *Rule* (*UncompressedFormattedMatch-to-match-expr m*) *a*. *r* ← *rs*]

lemma *format-Ln-rules-uncompressed-correct*: *good-ruleset rs* ⟹
approximating-bigstep-fun γ *p* (*Ln-rules-to-rule* (*format-Ln-rules-uncompressed rs*)) *s* = *approximating-bigstep-fun* γ *p rs s*
proof(*induction rs*)
case *Nil* **thus** *?case* **by**(*simp add*: *Ln-rules-to-rule-def format-Ln-rules-uncompressed-def*)
next
case (*Cons r rs*)
from *Cons.IH Cons.prems good-ruleset-tail* **have** *IH*: *approximating-bigstep-fun* γ *p* (*Ln-rules-to-rule* (*format-Ln-rules-uncompressed rs*)) *s* = *approximating-bigstep-fun* γ *p rs s*
by *blast*

have *map-uncompressedmapping-simp*: $\bigwedge$*ms a*. (*map* ((λ(*m*, *y*). *Rule* (*UncompressedFormattedMatch-to-match-expr m*) *y*) ∘ (λ*r*. (*format-Ln-match* (*get-match r*), *get-action r*)) ∘ (λ*m*. *Rule m a*)) *ms*) =
(*map* (λ*m*. *Rule m a*) (*map* (λ*m'*. *UncompressedFormattedMatch-to-match-expr* (*format-Ln-match m'*)) *ms*)) **by**(*simp*)

let *?rsA*=*map* ((λ(*m*, *y*). *Rule* (*UncompressedFormattedMatch-to-match-expr m*) *y*) ∘ (λ*r*. (*format-Ln-match* (*get-match r*), *get-action r*))) (*normalize-rules* [*r*])
let *?rsC*=*map* ((λ(*m*, *y*). *Rule* (*UncompressedFormattedMatch-to-match-expr m*) *y*) ∘ (λ*r*. (*format-Ln-match* (*get-match r*), *get-action r*))) (*normalize-rules rs*)
from *approximating-bigstep-fun-wf-postpend*[**where** *rsB*=[*r*], *simplified*] **have** *subst-rule*:
$\bigwedge$*rsA rsC*. *wf-ruleset* γ *p rsA* ⟹ *wf-ruleset* γ *p* [*r*] ⟹
approximating-bigstep-fun γ *p rsA s* = *approximating-bigstep-fun* γ *p* [*r*] *s* ⟹ *approximating-bigstep-fun* γ *p* (*rsA* @ *rsC*) *s* = *approximating-bigstep-fun* γ *p* (*r* # *rsC*) *s*
by *fast*

have *approximating-bigstep-fun* γ *p* (*Ln-rules-to-rule* (*format-Ln-rules-uncompressed* (*r* # *rs*))) *s* =

approximating-bigstep-fun γ *p*
(*map* ((λ(*m*, *y*). *Rule* (*UncompressedFormattedMatch-to-match-expr m*) *y*) ∘ (λ*r*. (*format-Ln-match* (*get-match r*), *get-action r*))) (*normalize-rules* (*r* # *rs*))) *s*
by (*simp add*: *Ln-rules-to-rule-def format-Ln-rules-uncompressed-def*)
also have ... = *approximating-bigstep-fun* γ *p* (*?rsA* @ *?rsC*) *s* **by**(*subst normalize-rules-fst*, *simp*)
also have ... = *approximating-bigstep-fun* γ *p* (*r* # *?rsC*) *s*
proof(*subst subst-rule*)
from *Cons.prems good-ruleset-fst* **have** *good-ruleset* [*r*] **by** *fast*
hence *good-ruleset ?rsA*
apply(*cases r*)
apply(*rename-tac m a*,*clarify*)
apply(*simp add*: *normalize-rules-singleton*)
apply(*drule good-ruleset-normalize-match*)
apply(*simp add*: *good-ruleset-alt*)
done
thus *wf-ruleset* γ *p ?rsA* **using** *good-imp-wf-ruleset* **by** *fast*
next
show *wf-ruleset* γ *p* [*r*] **using** *Cons.prems good-ruleset-fst good-imp-wf-ruleset* **by** *fast*
next
show *approximating-bigstep-fun* γ *p ?rsA s* = *approximating-bigstep-fun* γ *p* [*r*] *s*
apply(*case-tac r*, *rename-tac m a*, *clarify*)
apply(*simp add*: *normalize-rules-singleton*)
apply(*simp add*: *map-uncompressedmapping-simp*)
apply(*subst normalize-match-correct*[*symmetric*])
apply(*subst format-Ln-match-correct'*[*symmetric*])
apply(*simp add*: *normalized-match-normalize-match*)
apply(*simp*)
done
next
show *approximating-bigstep-fun* γ *p* (*r*#*?rsC*) *s* = *approximating-bigstep-fun* γ *p* (*r*#*?rsC*) *s* **by** *blast*
qed
also have ... = *approximating-bigstep-fun* γ *p* (*r* # *Ln-rules-to-rule* (*format-Ln-rules-uncompressed rs*)) *s*
by (*simp add*: *Ln-rules-to-rule-def format-Ln-rules-uncompressed-def*)
also have ... = *approximating-bigstep-fun* γ *p* (*r* # *rs*) *s* **using** *IH approximating-bigstep-fun-singleton-preper* **by** *fast*
finally show *?case* **.**
qed

fun *Ln-uncompressed-matching* :: (*iptrule-match*, *'packet*) *match-tac* ⇒ *action* ⇒ *'packet* ⇒ *match-Ln-uncompressed* ⇒ *bool* **where**
Ln-uncompressed-matching γ *a p* (*UncompressedFormattedMatch src dst proto extra*) ⟷
(*nt-match-list* γ *a p* (*NegPos-map Src src*)) ∧
(*nt-match-list* γ *a p* (*NegPos-map Dst dst*)) ∧
(*nt-match-list* γ *a p* (*NegPos-map Prot proto*)) ∧
(*nt-match-list* γ *a p* (*NegPos-map Extra extra*))

declare *Ln-uncompressed-matching.simps*[*simp del*]

lemma *Ln-uncompressed-matching*: *Ln-uncompressed-matching* γ *a p m* ⟷ *matches* γ (*UncompressedFormattedMatch-to-match-expr m*) *a p*
apply(*cases m*)
apply(*simp*)
apply(*simp add*: *nt-match-list-matches Ln-uncompressed-matching.simps*)
by (*metis matches-simp1 matches-simp2*)

lemma *Ln-uncompressed-matching-semantics-singleton*: *Ln-uncompressed-matching* γ *a p m1* ⟷ *Ln-uncompressed-matching* γ *a p m2*
⟹ *approximating-bigstep-fun* γ *p* (*Ln-rules-to-rule* [(*m1*, *a*)]) *s* =
approximating-bigstep-fun γ *p* (*Ln-rules-to-rule* [(*m2*, *a*)]) *s*
apply(*case-tac s*)
prefer *2*
apply(*simp add*: *Decision-approximating-bigstep-fun*)
apply(*clarify*)
apply(*simp add*: *Ln-rules-to-rule-def*)
apply(*simp split*: *action.split*)
apply(*simp add*: *Ln-uncompressed-matching*)
apply(*safe*)
done

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

20.1 Primitive Matchers: IP Space Matcher

fun *simple-matcher* :: (*iptrule-match*, *packet*) *exact-match-tac* **where**
simple-matcher (*Src* (*Ip4Addr ip*)) *p* = *bool-to-ternary* (*ipv4addr-of-dotteddecimal ip* = *src-ip p*) |
simple-matcher (*Src* (*Ip4AddrNetmask ip n*)) *p* = *bool-to-ternary* (*src-ip p* ∈ *ipv4range-set-from-bitmask* (*ipv4addr-of-dotteddecimal ip*) *n*) |

simple-matcher (*Dst* (*Ip4Addr ip*)) *p* = *bool-to-ternary* (*ipv4addr-of-dotteddecimal ip* = *dst-ip p*) |
simple-matcher (*Dst* (*Ip4AddrNetmask ip n*)) *p* = *bool-to-ternary* (*dst-ip p* ∈ *ipv4range-set-from-bitmask* (*ipv4addr-of-dotteddecimal ip*) *n*) |

simple-matcher (*Prot ProtAll*) - = *TernaryTrue* |
simple-matcher (*Prot ipt-protocol.ProtTCP*) *p* = *bool-to-ternary* (*prot p* = *prot-Packet.ProtTCP*) |
simple-matcher (*Prot ipt-protocol.ProtUDP*) *p* = *bool-to-ternary* (*prot p* = *prot-Packet.ProtUDP*) |

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

lemma *simple-matcher-simps*[*simp*]:
simple-matcher (*Src ip*) *p* = *bool-to-ternary* (*src-ip p* ∈ *ipv4s-to-set ip*)
simple-matcher (*Dst ip*) *p* = *bool-to-ternary* (*dst-ip p* ∈ *ipv4s-to-set ip*)
apply(*case-tac* [!] *ip*)
apply(*auto*)
apply (*metis* (*poly-guards-query*) *bool-to-ternary-simps*(*2*))+
done

declare *simple-matcher.simps*(*1*)[*simp del*]
declare *simple-matcher.simps*(*2*)[*simp del*]
declare *simple-matcher.simps*(*3*)[*simp del*]
declare *simple-matcher.simps*(*4*)[*simp del*]

Perform very basic optimizations

fun *opt-simple-matcher* :: *iptrule-match match-expr* ⇒ *iptrule-match match-expr*
where
opt-simple-matcher (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) = *MatchAny* |
opt-simple-matcher (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) = *MatchAny* |
opt-simple-matcher (*Match* (*Prot ProtAll*)) = *MatchAny* |
opt-simple-matcher (*Match m*) = *Match m* |
opt-simple-matcher (*MatchNot m*) = (*MatchNot* (*opt-simple-matcher m*)) |
opt-simple-matcher (*MatchAnd m1 m2*) = *MatchAnd* (*opt-simple-matcher m1*) (*opt-simple-matcher m2*) |
opt-simple-matcher MatchAny = *MatchAny*

lemma *opt-simple-matcher-correct-matchexpr*: *matches* (*simple-matcher*, α) *m* = *matches* (*simple-matcher*, α) (*opt-simple-matcher m*)
apply(*simp add*: *fun-eq-iff*, *clarify*, *rename-tac a p*)
apply(*rule matches-iff-apply-f*)
apply(*simp*)
apply(*induction m rule*: *opt-simple-matcher.induct*)

apply(*simp-all add*: *eval-ternary-simps ip-in-ipv4range-set-from-bitmask-UNIV*)
done
corollary *opt-simple-matcher-correct*: *approximating-bigstep-fun* (*simple-matcher*, α) *p* (*optimize-matches opt-simple-matcher rs*) *s* = *approximating-bigstep-fun* (*simple-matcher*, α) *p rs s*
using *optimize-matches opt-simple-matcher-correct-matchexpr* **by** *metis*

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

fun *opt-simple-matcher-in-doubt-allow-extra* :: *action* $\Rightarrow$ *iptrule-match match-expr* $\Rightarrow$ *iptrule-match match-expr* **where**
opt-simple-matcher-in-doubt-allow-extra - *MatchAny* = *MatchAny* |
opt-simple-matcher-in-doubt-allow-extra Accept (*Match* (*Extra* -)) = *MatchAny* |
opt-simple-matcher-in-doubt-allow-extra Reject (*Match* (*Extra* -)) = *MatchNot MatchAny* |
opt-simple-matcher-in-doubt-allow-extra Drop (*Match* (*Extra* -)) = *MatchNot MatchAny* |
opt-simple-matcher-in-doubt-allow-extra - (*Match m*) = *Match m* |
opt-simple-matcher-in-doubt-allow-extra Accept (*MatchNot* (*Match* (*Extra* -))) = *MatchAny* |
opt-simple-matcher-in-doubt-allow-extra Drop (*MatchNot* (*Match* (*Extra* -))) = *MatchNot MatchAny* |
opt-simple-matcher-in-doubt-allow-extra Reject (*MatchNot* (*Match* (*Extra* -))) = *MatchNot MatchAny* |
opt-simple-matcher-in-doubt-allow-extra a (*MatchNot* (*MatchNot m*)) = *opt-simple-matcher-in-doubt-allow-ex*
a m |

— $\neg$ ($a \wedge b$) = $\neg$ $b \vee \neg$ a and $\neg$ *Unknown* = *Unknown*
opt-simple-matcher-in-doubt-allow-extra a (*MatchNot* (*MatchAnd m1 m2*)) =
(*if* (*opt-simple-matcher-in-doubt-allow-extra a* (*MatchNot m1*)) = *MatchAny* $\vee$
(*opt-simple-matcher-in-doubt-allow-extra a* (*MatchNot m2*)) = *MatchAny*
then MatchAny else
(*if* (*opt-simple-matcher-in-doubt-allow-extra a* (*MatchNot m1*)) = *MatchNot MatchAny then*
opt-simple-matcher-in-doubt-allow-extra a (*MatchNot m2*) *else*
if (*opt-simple-matcher-in-doubt-allow-extra a* (*MatchNot m2*)) = *MatchNot MatchAny then*
opt-simple-matcher-in-doubt-allow-extra a (*MatchNot m1*) *else*
MatchNot (*MatchAnd* (*MatchNot* (*opt-simple-matcher-in-doubt-allow-extra a* (*MatchNot m1*))) (*MatchNot* (*opt-simple-matcher-in-doubt-allow-extra a* (*MatchNot m2*)))))
) |
— For the final else case, we exploit idempotence of not to introduce recursive calls

opt-simple-matcher-in-doubt-allow-extra - (*MatchNot m*) = *MatchNot m* |
opt-simple-matcher-in-doubt-allow-extra a (*MatchAnd m1 m2*) = *MatchAnd* (*opt-simple-matcher-in-doubt-all*
a m1) (*opt-simple-matcher-in-doubt-allow-extra a m2*)

lemma[*code-unfold*]: *opt-simple-matcher-in-doubt-allow-extra a* (*MatchNot* (*MatchAnd m1 m2*)) =
(*let m1′* = *opt-simple-matcher-in-doubt-allow-extra a* (*MatchNot m1*); *m2′* = *opt-simple-matcher-in-doubt-allow-extra 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′*)))
)
by(*simp*)

opt-simple-matcher-in-doubt-allow-extra can be expressed in terms of *remove-unknowns-generic*

lemma assumes $a = Accept \lor a = Drop$ **shows** *opt-simple-matcher-in-doubt-allow-extra a m* = *remove-unknowns-generic* (*simple-matcher*, *in-doubt-allow*) *a m*
proof −
{**fix** *x1 x2 x3*
have
(∃ *p*::*packet*. *bool-to-ternary* (*src-ip p* ∈ *ipv4s-to-set x1*) ≠ *TernaryUnknown*)
(∃ *p*::*packet*. *bool-to-ternary* (*dst-ip p* ∈ *ipv4s-to-set x2*) ≠ *TernaryUnknown*)
(∃ *p*::*packet*. *simple-matcher* (*Prot x3*) *p* ≠ *TernaryUnknown*)
apply −
apply(*simp-all add*: *bool-to-ternary-Unknown*)
apply(*case-tac x3*)
apply(*simp-all*)
apply(*rule-tac x*=⦇ *src-ip* = *X*, *dst-ip* = *Y*, *prot* = *protPacket.ProtTCP* ⦈ **in** *exI*, *simp*)
apply(*rule-tac x*=⦇ *src-ip* = *X*, *dst-ip* = *Y*, *prot* = *protPacket.ProtUDP* ⦈ **in** *exI*, *simp*)
done
} **note** *simple-matcher-packet-exists*=*this*
{**fix** γ
have *a* = *Accept* ∨ *a* = *Drop* ⟹ γ = (*simple-matcher*, *in-doubt-allow*) ⟹ *opt-simple-matcher-in-doubt-allow-extra a m* = *remove-unknowns-generic* γ *a m*
apply(*induction* γ *a m rule*: *remove-unknowns-generic.induct*)
apply(*simp-all*)
apply(*case-tac* [!] *A*)
apply(*case-tac* [!] *a*)
apply(*simp-all*)
apply(*simp-all add*: *simple-matcher-packet-exists*)
done
} **thus** *?thesis* **using** ⟨*a* = *Accept* ∨ *a* = *Drop*⟩ **by** *simp*
qed

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

lemma *simple-matcher-prot-not-unkown*: *simple-matcher* (*Prot v*) *p* $\neq$ *TernaryUnknown*
apply(*cases v*)
apply(*simp-all add*: *bool-to-ternary-Unknown*)
done

opt-simple-matcher-in-doubt-allow-extra does indeed remove all unknowns

theorem *opt-simple-matcher-in-doubt-allow-extra-specification*:
a = *Accept* $\vee$ *a* = *Drop* $\vee$ *a* = *Reject* $\Longrightarrow$ $\neg$ *has-unknowns simple-matcher* (*opt-simple-matcher-in-doubt-allow-extra a m*)
apply(*induction a m rule*: *opt-simple-matcher-in-doubt-allow-extra.induct*)
apply(*simp-all add*: *bool-to-ternary-Unknown simple-matcher-prot-not-unkown*)
done

value *opt-simple-matcher-in-doubt-allow-extra Drop* (*MatchNot* (*MatchAnd* (*MatchNot MatchAny*) (*MatchNot* (*MatchAnd* (*MatchNot MatchAny*) (*Match* (*Extra ''foo''*))))))
value *opt-simple-matcher-in-doubt-allow-extra Accept* ((*MatchAnd* (*MatchNot MatchAny*) (*MatchNot* (*MatchAnd* (*MatchNot MatchAny*) (*Match* (*Extra ''foo''*))))))

lemma *eval-ternary-And-UnknownTrue1*: *eval-ternary-And TernaryUnknown t* $\neq$ *TernaryTrue*
apply(*cases t*)
apply(*simp-all*)
done

lemma *matches* γ *m1 a p* = *matches* γ *m2 a p* $\Longrightarrow$ *matches* γ (*MatchNot m1*) *a p* = *matches* γ (*MatchNot m2*) *a p*
apply(*case-tac* γ)
apply(*simp add*: *matches-case-ternaryvalue-tuple split*:)
— counterexample: *m1* is unknown *m2* is true default matches
oops

lemma *opt-simple-matcher-in-doubt-allow-extra-correct-matchexpr*: *matches* (*simple-matcher*, *in-doubt-allow*) (*opt-simple-matcher-in-doubt-allow-extra a m*) *a* =
matches (*simple-matcher*, *in-doubt-allow*) *m a*
apply(*simp add*: *fun-eq-iff*, *clarify*)

apply(*rename-tac p*)
apply(*induction a m rule*: *opt-simple-matcher-in-doubt-allow-extra.induct*)
apply(*simp-all add*: *bunch-of-lemmata-about-matches matches-DeMorgan*)
apply(*simp-all add*: *matches-case-ternaryvalue-tuple*)

apply *safe*
apply(*simp-all*)
done

corollary *opt-simple-matcher-in-doubt-allow-extra-correct*:
approximating-bigstep-fun (*simple-matcher*, *in-doubt-allow*) *p* (*optimize-matches-a opt-simple-matcher-in-doubt-allow-extra rs*) *s* =
approximating-bigstep-fun (*simple-matcher*, *in-doubt-allow*) *p rs s*
using *optimize-matches-a opt-simple-matcher-in-doubt-allow-extra-correct-matchexpr*
by *metis*

fun *opt-simple-matcher-in-doubt-deny-extra* :: *action* ⇒ *iptrule-match match-expr* ⇒ *iptrule-match match-expr* **where**
opt-simple-matcher-in-doubt-deny-extra - *MatchAny* = *MatchAny* |
opt-simple-matcher-in-doubt-deny-extra Accept (*Match* (*Extra* -)) = *MatchNot MatchAny* |
opt-simple-matcher-in-doubt-deny-extra Reject (*Match* (*Extra* -)) = *MatchAny* |
opt-simple-matcher-in-doubt-deny-extra Drop (*Match* (*Extra* -)) = *MatchAny* |
opt-simple-matcher-in-doubt-deny-extra - (*Match m*) = *Match m* |
opt-simple-matcher-in-doubt-deny-extra Reject (*MatchNot* (*Match* (*Extra* -))) = *MatchAny* |
opt-simple-matcher-in-doubt-deny-extra Drop (*MatchNot* (*Match* (*Extra* -))) = *MatchAny* |
opt-simple-matcher-in-doubt-deny-extra Accept (*MatchNot* (*Match* (*Extra* -))) = *MatchNot MatchAny* |
opt-simple-matcher-in-doubt-deny-extra a (*MatchNot* (*MatchNot m*)) = *opt-simple-matcher-in-doubt-deny-ext a m* |

opt-simple-matcher-in-doubt-deny-extra a (*MatchNot* (*MatchAnd m1 m2*)) =
(*if* (*opt-simple-matcher-in-doubt-deny-extra a* (*MatchNot m1*)) = *MatchAny* ∨
(*opt-simple-matcher-in-doubt-deny-extra a* (*MatchNot m2*)) = *MatchAny*
then MatchAny else
(*if* (*opt-simple-matcher-in-doubt-deny-extra a* (*MatchNot m1*)) = *MatchNot MatchAny then*
opt-simple-matcher-in-doubt-deny-extra a (*MatchNot m2*) *else*
if (*opt-simple-matcher-in-doubt-deny-extra a* (*MatchNot m2*)) = *MatchNot MatchAny then*
opt-simple-matcher-in-doubt-deny-extra a (*MatchNot m1*) *else*
MatchNot (*MatchAnd* (*MatchNot* (*opt-simple-matcher-in-doubt-deny-extra a* (*MatchNot m1*))) (*MatchNot* (*opt-simple-matcher-in-doubt-deny-extra a* (*MatchNot m2*)))))

) |
opt-simple-matcher-in-doubt-deny-extra - (*MatchNot m*) = *MatchNot m* |
opt-simple-matcher-in-doubt-deny-extra a (*MatchAnd m1 m2*) = *MatchAnd* (*opt-simple-matcher-in-doubt-deny-extra a m1*) (*opt-simple-matcher-in-doubt-deny-extra a m2*)

lemma *opt-simple-matcher-in-doubt-deny-extra-correct-matchexpr*: *matches* (*simple-matcher*, *in-doubt-deny*) (*opt-simple-matcher-in-doubt-deny-extra a m*) *a* = *matches* (*simple-matcher*, *in-doubt-deny*) *m a*
apply(*simp add*: *fun-eq-iff*, *clarify*)
apply(*rename-tac p*)
apply(*induction a m rule*: *opt-simple-matcher-in-doubt-deny-extra.induct*)
apply(*simp-all add*: *bunch-of-lemmata-about-matches matches-DeMorgan*)
apply(*simp-all add*: *matches-case-ternaryvalue-tuple*)

apply *safe*
apply(*simp-all*)
done

corollary *opt-simple-matcher-in-doubt-deny-extra-correct*: *approximating-bigstep-fun* (*simple-matcher*, *in-doubt-deny*) *p* (*optimize-matches-a opt-simple-matcher-in-doubt-deny-extra rs*) *s* = *approximating-bigstep-fun* (*simple-matcher*, *in-doubt-deny*) *p rs s*
using *optimize-matches-a opt-simple-matcher-in-doubt-deny-extra-correct-matchexpr*
by *metis*

theorem *opt-simple-matcher-in-doubt-deny-extra-specification*:
a = *Accept* ∨ *a* = *Drop* ∨ *a* = *Reject* ⟹ ¬ *has-unknowns simple-matcher* (*opt-simple-matcher-in-doubt-deny-extra a m*)
apply(*induction a m rule*: *opt-simple-matcher-in-doubt-deny-extra.induct*)
apply(*simp-all add*: *bool-to-ternary-Unknown simple-matcher-prot-not-unkown*)
done

Lemmas when matching on *Src* or *Dst*

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

matches (*simple-matcher*, α) (*Match* (*Src X*)) *a p* $\longleftrightarrow$ *src-ip p* $\in$ *ipv4s-to-set X*
matches (*simple-matcher*, α) (*Match* (*Dst X*)) *a p* $\longleftrightarrow$ *dst-ip 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 (*simple-matcher*, α) (*MatchNot* (*Match* (*Src X*))) *a p* $\longleftrightarrow$ *src-ip p* $\notin$ *ipv4s-to-set X*
matches (*simple-matcher*, α) (*MatchNot* (*Match* (*Dst X*))) *a p* $\longleftrightarrow$ *dst-ip 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 *simple-matcher-SrcDst-Inter*:
($\forall$ *m*$\in$*set X*. *matches* (*simple-matcher*, α) (*Match* (*Src m*)) *a p*) $\longleftrightarrow$ *src-ip p* $\in$ ($\bigcap$ *x*$\in$*set X*. *ipv4s-to-set x*)
($\forall$ *m*$\in$*set X*. *matches* (*simple-matcher*, α) (*Match* (*Dst m*)) *a p*) $\longleftrightarrow$ *dst-ip p* $\in$ ($\bigcap$ *x*$\in$*set X*. *ipv4s-to-set x*)
apply(*simp-all*)
apply(*simp-all add*: *matches-case-ternaryvalue-tuple bool-to-ternary-Unknown bool-to-ternary-simps split*: *ternaryvalue.split*)
done

end
theory *IPSpace-Format-Ln*
imports *Format-Ln ../Primitive-Matchers/IPSpace-Matcher IPSpace-Operations*
begin

20.2 Formatting

lemma ($\bigcap$ *x*$\in$*set X*. *ipv4s-to-set x*) = {} $\Longrightarrow$ $\neg$ ($\forall$ *m*$\in$*set X*. *matches* (*simple-matcher*, α) (*Match* (*Src m*)) *a p*)
using *simple-matcher-SrcDst-Inter* **by** *blast*

lemma *compress-pos-ips-src-None-matching*: *compress-pos-ips src'* = *None* $\Longrightarrow$
$\neg$ *Ln-uncompressed-matching* (*simple-matcher*, α) *a p* (*UncompressedFormattedMatch* (*map Pos src'*) *dst proto extra*)
apply(*simp add*: *compress-pos-ips-None*)
apply(*unfold Ln-uncompressed-matching.simps*)
apply *safe*
apply(*thin-tac nt-match-list* (*simple-matcher*, α) *a p* (*NegPos-map Dst dst*))
apply(*thin-tac nt-match-list* (*simple-matcher*, α) *a p* (*NegPos-map Prot proto*))
apply(*thin-tac nt-match-list* (*simple-matcher*, α) *a p* (*NegPos-map Extra extra*))
apply(*simp add*: *nt-match-list-simp*)
apply(*simp add*: *getPos-NegPos-map-simp*)

using *simple-matcher-SrcDst-Inter* **by** *blast*
lemma *compress-pos-ips-dst-None-matching*: *compress-pos-ips dst* = *None* $\Longrightarrow$
$\neg$ *Ln-uncompressed-matching* (*simple-matcher*, α) *a p* (*UncompressedFormattedMatch src* (*map Pos dst*) *proto extra*)
apply(*simp add*: *compress-pos-ips-None*)
apply(*unfold Ln-uncompressed-matching.simps*)
apply *safe*
apply(*thin-tac nt-match-list* (*simple-matcher*, α) *a p* (*NegPos-map Src ?x*))
apply(*thin-tac nt-match-list* (*simple-matcher*, α) *a p* (*NegPos-map Prot proto*))
apply(*thin-tac nt-match-list* (*simple-matcher*, α) *a p* (*NegPos-map Extra extra*))
apply(*simp add*: *nt-match-list-simp*)
apply(*simp add*: *getPos-NegPos-map-simp*)
using *simple-matcher-SrcDst-Inter* **by** *blast*

lemma *compress-pos-ips-src-Some-matching*: *compress-pos-ips src'* = *Some X* $\Longrightarrow$

matches (*simple-matcher*, α) (*alist-and* (*NegPos-map Src* [*Pos X*])) *a p* $\longleftrightarrow$
matches (*simple-matcher*, α) (*alist-and* (*NegPos-map Src* (*map Pos src'*)))*a p*
apply(*drule compress-pos-ips-Some*)
apply(*simp only*: *nt-match-list-matches*[*symmetric*])
apply *safe*
apply(*simp add*: *nt-match-list-simp*)
apply(*simp add*: *getPos-NegPos-map-simp*)
apply(*rule conjI*)
apply(*simp add*: *simple-matcher-SrcDst-Inter*)
apply(*simp add*: *match-simplematcher-SrcDst*)
apply(*simp add*: *getNeg-Pos-empty*)
apply(*simp add*: *match-simplematcher-SrcDst*)
apply(*simp add*: *nt-match-list-simp*)
apply(*simp add*: *getPos-NegPos-map-simp*)
apply(*simp add*: *simple-matcher-SrcDst-Inter*)
done
lemma *compress-pos-ips-dst-Some-matching*: *compress-pos-ips dst'* = *Some X* $\Longrightarrow$

matches (*simple-matcher*, α) (*alist-and* (*NegPos-map Dst* [*Pos X*])) *a p* $\longleftrightarrow$
matches (*simple-matcher*, α) (*alist-and* (*NegPos-map Dst* (*map Pos dst'*)))*a p*
apply(*drule compress-pos-ips-Some*)
apply(*simp only*: *nt-match-list-matches*[*symmetric*])
apply *safe*
apply(*simp add*: *nt-match-list-simp*)
apply(*simp add*: *getPos-NegPos-map-simp*)
apply(*rule conjI*)
apply(*simp add*: *simple-matcher-SrcDst-Inter*)
apply(*simp add*: *match-simplematcher-SrcDst*)
apply(*simp add*: *getNeg-Pos-empty*)
apply(*simp add*: *match-simplematcher-SrcDst*)
apply(*simp add*: *nt-match-list-simp*)
apply(*simp add*: *getPos-NegPos-map-simp*)

apply(*simp add*: *simple-matcher-SrcDst-Inter*)
done

lemma *Ln-uncompressed-matching-src-dst-subset*: *set* (*src'*) $\subseteq$ *set* (*src*) $\Longrightarrow$
Ln-uncompressed-matching (*simple-matcher*, α) *a p* (*UncompressedFormattedMatch src dst proto extra*) $\Longrightarrow$
Ln-uncompressed-matching (*simple-matcher*, α) *a p* (*UncompressedFormattedMatch src' dst proto extra*)
set (*dst'*) $\subseteq$ *set* (*dst*) $\Longrightarrow$
Ln-uncompressed-matching (*simple-matcher*, α) *a p* (*UncompressedFormattedMatch src dst proto extra*) $\Longrightarrow$
Ln-uncompressed-matching (*simple-matcher*, α) *a p* (*UncompressedFormattedMatch src dst' proto extra*)
apply(*simp-all only*: *Ln-uncompressed-matching.simps nt-match-list-matches*)
apply(*safe*)
apply(*thin-tac matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Dst ?x*)) *a p*)
apply(*thin-tac matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Prot ?x*)) *a p*)
apply(*thin-tac matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Extra ?x*)) *a p*)
prefer *2*
apply(*thin-tac matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Src ?x*)) *a p*)
apply(*thin-tac matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Prot ?x*)) *a p*)
apply(*thin-tac matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Extra ?x*)) *a p*)
prefer *2*
apply(*simp-all add*: *matches-alist-and*)
apply(*simp-all add*: *NegPos-map-simps*)
apply(*simp-all add*: *match-simplematcher-SrcDst match-simplematcher-SrcDst-not*)
apply(*clarify*)
apply(*simp-all add*: *NegPos-set*)
apply *blast*
apply(*clarify*)
apply(*blast*)
done

lemma *compress-ips-src-None-matching*: *compress-ips src* = *None* $\Longrightarrow$ $\neg$ *Ln-uncompressed-matching* (*simple-matcher*, α) *a p* (*UncompressedFormattedMatch src dst proto extra*)
apply(*case-tac getPos src* = [])
apply(*simp*)
apply(*simp split*: *option.split-asm*)

apply(*drule-tac* α=α **and** *a*=*a* **and** *p*=*p* **and** *dst*=*dst* **and** *proto*=*proto* **and** *extra*=*extra* **in** *compress-pos-ips-src-None-matching*)
apply(*thin-tac getPos src* $\neq$ [])
apply(*erule HOL.rev-notE*)
apply(*simp*)
apply(*rule-tac src'*=(*map Pos* (*getPos src*)) **and** *src*=*src* **in** *Ln-uncompressed-matching-src-dst-subset*(*1*))
prefer *2* **apply** *simp*
apply(*simp*)
apply(*simp add*: *NegPos-set*)
apply(*simp split*: *split-if-asm*)
apply(*drule compress-pos-ips-Some*)
apply(*simp add*: *ipv4range-to-set-collect-to-range ipv4range-set-from-bitmask-to-executable-ipv4range*)
apply(*simp add*: *Ln-uncompressed-matching.simps nt-match-list-matches*)
apply(*clarify*)
apply(*thin-tac matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Dst ?x*)) *a* *p*)
apply(*thin-tac matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Prot ?x*)) *a p*)
apply(*thin-tac matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Extra ?x*)) *a p*)
apply(*simp add*: *matches-alist-and*)
apply(*simp add*: *NegPos-map-simps*)
apply(*simp add*: *match-simplematcher-SrcDst match-simplematcher-SrcDst-not*)
apply(*clarify*)
by (*metis* (*erased*, *hide-lams*) *INT-iff UN-iff subsetCE*)
lemma *compress-ips-dst-None-matching*: *compress-ips dst* = *None* $\Longrightarrow$ $\neg$ *Ln-uncompressed-matching* (*simple-matcher*, α) *a p* (*UncompressedFormattedMatch src dst proto extra*)
apply(*case-tac getPos dst* = [])
apply(*simp*)
apply(*simp split*: *option.split-asm*)
apply(*drule-tac* α=α **and** *a*=*a* **and** *p*=*p* **and** *src*=*src* **and** *proto*=*proto* **and** *extra*=*extra* **in** *compress-pos-ips-dst-None-matching*)
apply(*thin-tac getPos dst* $\neq$ [])
apply(*erule HOL.rev-notE*)
apply(*simp*)
apply(*rule-tac dst'*=(*map Pos* (*getPos dst*)) **and** *dst*=*dst* **in** *Ln-uncompressed-matching-src-dst-subset*(*2*))
prefer *2* **apply** *simp*
apply(*simp*)
apply(*simp add*: *NegPos-set*)
apply(*simp split*: *split-if-asm*)
apply(*drule compress-pos-ips-Some*)
apply(*simp add*: *ipv4range-to-set-collect-to-range ipv4range-set-from-bitmask-to-executable-ipv4range*)
apply(*simp add*: *Ln-uncompressed-matching.simps nt-match-list-matches*)
apply(*clarify*)
apply(*thin-tac matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Src ?x*)) *a* *p*)
apply(*thin-tac matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Prot ?x*)) *a p*)
apply(*thin-tac matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Extra ?x*))

a p)
apply(*simp add*: *matches-alist-and*)
apply(*simp add*: *NegPos-map-simps*)
apply(*simp add*: *match-simplematcher-SrcDst match-simplematcher-SrcDst-not*)
apply(*clarify*)
by (*metis* (*erased*, *hide-lams*) *INT-iff UN-iff subsetCE*)

lemma *Ln-uncompressed-matching-src-eq*: *matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Src X*)) *a p* $\longleftrightarrow$ *matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Src Y*)) *a p* $\Longrightarrow$
Ln-uncompressed-matching (*simple-matcher*, α) *a p* (*UncompressedFormattedMatch X dst proto extra*) $\longleftrightarrow$
Ln-uncompressed-matching (*simple-matcher*, α) *a p* (*UncompressedFormattedMatch Y dst proto extra*)
apply(*simp add*: *Ln-uncompressed-matching*)
by (*metis matches-simp11 matches-simp22*)

lemma *Ln-uncompressed-matching-src-dst-eq*: *matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Src X*)) *a p* $\longleftrightarrow$ *matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Src Y*)) *a p* $\Longrightarrow$
matches (*simple-matcher*, α) (*alist-and* (*NegPos-map Dst A*)) *a p* $\longleftrightarrow$ *matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Dst B*)) *a p* $\Longrightarrow$
Ln-uncompressed-matching (*simple-matcher*, α) *a p* (*UncompressedFormattedMatch X A proto extra*) $\longleftrightarrow$
Ln-uncompressed-matching (*simple-matcher*, α) *a p* (*UncompressedFormattedMatch Y B proto extra*)
apply(*simp add*: *Ln-uncompressed-matching*)
by (*metis matches-simp11 matches-simp22*)

lemma *matches-and-x-any*: *matches* γ (*MatchAnd* (*Match x*) *MatchAny*) *a p* = *matches* γ (*Match x*) *a p*
apply(*case-tac* γ)
by(*simp add*: *matches-case-ternaryvalue-tuple split*: *ternaryvalue.split*)

lemma *compress-ips-src-Some-matching*: *compress-ips src* = *Some X* $\Longrightarrow$
matches (*simple-matcher*, α) (*alist-and* (*NegPos-map Src X*)) *a p* $\longleftrightarrow$ *matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Src src*)) *a p*
apply(*case-tac getPos src* = [])
apply(*simp*)
apply(*simp*)
apply(*simp split*: *option.split-asm split-if-asm*)
apply(*simp add*: *ipv4range-set-from-bitmask-to-executable-ipv4range ipv4range-to-set-collect-to-range*)
apply(*drule-tac* α=α **and** *a*=*a* **and** *p*=*p* **in** *compress-pos-ips-src-Some-matching*)
apply(*simp add*: *matches-and-x-any*)
apply(*simp add*: *matches-alist-and NegPos-map-simps match-simplematcher-SrcDst match-simplematcher-SrcDst-not*)

apply(*safe*)
apply(*simp-all add*: *NegPos-map-simps*)
done

lemma *compress-ips-dst-Some-matching*: *compress-ips dst* = *Some X* $\Longrightarrow$
matches (*simple-matcher*, α) (*alist-and* (*NegPos-map Dst X*)) *a p* $\longleftrightarrow$ *matches* (*simple-matcher*, α) (*alist-and* (*NegPos-map Dst dst*)) *a p*
apply(*case-tac getPos dst* = [])
apply(*simp*)
apply(*simp*)
apply(*simp split*: *option.split-asm split-if-asm*)
apply(*simp add*: *ipv4range-set-from-bitmask-to-executable-ipv4range ipv4range-to-set-collect-to-range*)
apply(*drule-tac* α=α **and** *a*=*a* **and** *p*=*p* **in** *compress-pos-ips-dst-Some-matching*)
apply(*simp add*: *matches-and-x-any*)
apply(*simp add*: *matches-alist-and NegPos-map-simps match-simplematcher-SrcDst match-simplematcher-SrcDst-not*)
apply(*safe*)
apply(*simp-all add*: *NegPos-map-simps*)
done

fun *compress-Ln-ips* :: (*match-Ln-uncompressed* × *action*) *list* ⇒ (*match-Ln-uncompressed* × *action*) *list* **where**
compress-Ln-ips [] = [] |
compress-Ln-ips (((*UncompressedFormattedMatch src dst proto extra*), *a*)#*rs*) =
(*case* (*compress-ips src*, *compress-ips dst*) *of*
(*None*, -) ⇒ *compress-Ln-ips rs*
| (-, *None*) ⇒ *compress-Ln-ips rs*
| (*Some src'*, *Some dst'*) ⇒ (*UncompressedFormattedMatch src' dst' proto extra*, *a*)#(*compress-Ln-ips rs*)
)

export-code *compress-Ln-ips* **in** *SML*

fun *compress-prots* :: *ipt-protocol negation-type list* ⇒ *ipt-protocol negation-type option* **where**
compress-prots [] = *Some* (*Pos ProtAll*) |
compress-prots ((*Pos ProtAll*)#*ps*) = *compress-prots ps* |
compress-prots ((*Neg ProtAll*)#-) = *None* |
compress-prots (*p* # *Pos ProtAll* # *ps*) = *compress-prots* (*p*#*ps*)|
compress-prots (- # *Neg ProtAll* # -) = *None* |
compress-prots (*Pos ProtTCP* # *Pos ProtUDP* # -) = *None*|
compress-prots (*Pos ProtUDP* # *Pos ProtTCP* # -) = *None*

lemma *approximating-bigstep-fun-Ln-rules-to-rule-step-simultaneously*:
approximating-bigstep-fun (*simple-matcher*, α) *p* (*Ln-rules-to-rule* (*rs1*)) *Undecided* = *approximating-bigstep-fun* (*simple-matcher*, α) *p* (*Ln-rules-to-rule* (*rs2*))

Undecided ⟹
matches (*simple-matcher*, α) (*UncompressedFormattedMatch-to-match-expr r1*) *a p* ⟷ *matches* (*simple-matcher*, α) (*UncompressedFormattedMatch-to-match-expr r2*) *a p*
⟹
approximating-bigstep-fun (*simple-matcher*, α) *p* (*Ln-rules-to-rule* ((*r1*, *a*)#*rs1*)) *Undecided* =
approximating-bigstep-fun (*simple-matcher*, α) *p* (*Ln-rules-to-rule* ((*r2*, *a*)#*rs2*)) *Undecided*
by(*simp add*: *Ln-rules-to-rule-def split*: *action.split*)

theorem *compress-Ln-ips-xorrectness*: *approximating-bigstep-fun* (*simple-matcher*, α) *p* (*Ln-rules-to-rule* (*compress-Ln-ips rs1*)) *s* =
approximating-bigstep-fun (*simple-matcher*, α) *p* (*Ln-rules-to-rule rs1*) *s*
apply(*case-tac s*)
prefer *2*
apply(*simp add*: *Decision-approximating-bigstep-fun*)
apply(*clarify*, *thin-tac s* = *Undecided*)
apply(*induction rs1*)
apply(*simp*)
apply(*rename-tac r rs*)
apply(*case-tac r*, *simp*)
apply(*rename-tac m action*)
apply(*case-tac m*)
apply(*rename-tac src dst proto extra*)
apply(*simp only*:*compress-Ln-ips.simps*)
apply(*simp del*: *compress-ips.simps split*: *option.split*)
apply(*safe*)
apply(*drule-tac* α=α **and** *p*=*p* **and** *proto*=*proto* **and** *extra*=*extra* **and** *dst*=*dst* **and** *a*=*action* **in** *compress-ips-src-None-matching*)
apply(*simp add*: *Ln-rules-to-rule-def Ln-uncompressed-matching*)
apply(*drule-tac* α=α **and** *p*=*p* **and** *proto*=*proto* **and** *extra*=*extra* **and** *src*=*src* **and** *a*=*action* **in** *compress-ips-dst-None-matching*)
apply(*simp add*: *Ln-rules-to-rule-def Ln-uncompressed-matching*)
apply(*simp del*: *compress-ips.simps*)
apply(*drule-tac* α=α **and** *p*=*p* **and** *a*=*action* **in** *compress-ips-dst-Some-matching*)

apply(*drule-tac* α=α **and** *p*=*p* **and** *a*=*action* **in** *compress-ips-src-Some-matching*)
apply(*rule approximating-bigstep-fun-Ln-rules-to-rule-step-simultaneously*, *simp*)
apply(*rule Ln-uncompressed-matching-src-dst-eq*[*simplified Ln-uncompressed-matching*])
apply(*simp-all*)
done

fun *does-I-has-compressed-rules* :: (*match-Ln-uncompressed* × *action*) *list* ⇒ (*match-Ln-uncompressed* × *action*) *list* **where**
does-I-has-compressed-rules [] = [] |
does-I-has-compressed-rules (((*UncompressedFormattedMatch* [] [*dst*] *proto* []), *a*)#*rs*) =

does-I-has-compressed-rules rs |
does-I-has-compressed-rules (((*UncompressedFormattedMatch* [*src*] [] *proto* []), *a*)#*rs*) =
does-I-has-compressed-rules rs |
does-I-has-compressed-rules (((*UncompressedFormattedMatch* [*src*] [*dst*] *proto* []), *a*)#*rs*) =
does-I-has-compressed-rules rs |
does-I-has-compressed-rules (((*UncompressedFormattedMatch* [] [] *proto* []), *a*)#*rs*) =
does-I-has-compressed-rules rs |
does-I-has-compressed-rules (*r*#*rs*) = *r* # *does-I-has-compressed-rules rs*

fun *does-I-has-compressed-prots* :: (*match-Ln-uncompressed* × *action*) *list* ⇒ (*match-Ln-uncompressed* × *action*) *list* **where**
does-I-has-compressed-prots [] = [] |
does-I-has-compressed-prots (((*UncompressedFormattedMatch src dst* [] []), *a*)#*rs*) =
does-I-has-compressed-prots rs|
does-I-has-compressed-prots (((*UncompressedFormattedMatch src dst* [*proto*] []), *a*)#*rs*) =
does-I-has-compressed-prots rs|
does-I-has-compressed-prots (*r*#*rs*) =
r # *does-I-has-compressed-prots rs*

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

21 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.

21.1 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 = {$p \in P$. *matches* γ *m Accept p*} ∪ (*collect-allow* γ *rs* {$p \in P$. ¬ *matches* γ *m Accept p*}) |
collect-allow γ ((*Rule m Drop*)#*rs*) P = (*collect-allow* γ *rs* {$p \in P$. ¬ *matches* γ *m Drop p*})

lemma *collect-allow-subset*: *simple-ruleset rs* ⟹ *collect-allow* γ *rs* $P \subseteq P$

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* $\vee$ *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 FinalAllow* $\longrightarrow p \in P \longrightarrow p \in$ *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* $\Longrightarrow$ $\{p.\ p \in$ *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* $\Rightarrow$ '*a rule list* $\Rightarrow$ '*p set* $\Rightarrow$ '*p set* **where**
collect-allow-compl - [] *P* = *UNIV* |
collect-allow-compl γ ((*Rule m Accept*)#*rs*) *P* = (*P* $\cup$ $\{p.\ \neg$*matches* γ *m Accept p*$\}$) $\cap$ (*collect-allow-compl* γ *rs* (*P* $\cup$ $\{p.$ *matches* γ *m Accept p*$\}$)) |
collect-allow-compl γ ((*Rule m Drop*)#*rs*) *P* = (*collect-allow-compl* γ *rs* (*P* $\cup$

$\{p.\ matches\ \gamma\ m\ Drop\ p\}))$

lemma *collect-allow-compl-correct*: *simple-ruleset rs* $\Longrightarrow$ $(-$ *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*: $-\ \{p \in P.\ \neg$ *matches* γ *r Accept p*$\}$ = $-\ P \cup \{p.$ *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=$\{p \in P.\ \neg$ *matches* γ *r Accept p*$\}$] *set-simp1* **have**
$-$ *collect-allow-compl* γ *rs* $(-\ P \cup$ *Collect* (*matches* γ *r Accept*)) = *collect-allow* γ *rs* $\{p \in P.\ \neg$ *matches* γ *r Accept p*$\}$ **by** *simp*
thus *?case* **by** *auto*
next
case (*3* 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 *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=$\{p \in P.\ \neg$ *matches* γ *r Drop p*$\}$] *set-simp1* **have**
$-$ *collect-allow-compl* γ *rs* $(-\ P \cup$ *Collect* (*matches* γ *r Drop*)) = *collect-allow* γ *rs* $\{p \in P.\ \neg$ *matches* γ *r Drop p*$\}$ **by** *simp*
thus *?case* **by** *auto*
qed(*simp-all add*: *simple-ruleset-def*)

21.2 The set of all dropped packets

Collect all packets which are denied by the firewall.

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

lemma *collect-deny-subset*: *simple-ruleset rs* $\Longrightarrow$ *collect-deny* γ *rs P* $\subseteq$ *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* $\Longrightarrow$ $p \in$ *collect-deny* γ *rs P* $\Longrightarrow$ *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* $\vee$ *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** ($p \in P \wedge$ *matches* γ *m Drop p*) $\vee$ $p \in$ *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* ∨ *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 FinalDeny* ⟶ *p* ∈ *P* ⟶ *p* ∈ *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* ⟹ {*p*. *p* ∈ *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* ⇒ '*a rule list* ⇒ '*p set* ⇒ '*p set* **where**
collect-deny-compl - [] *P* = *UNIV* |
collect-deny-compl γ ((*Rule m Drop*)#*rs*) *P* = (*P* ∪ {*p*. ¬*matches* γ *m Drop p*}) ∩ (*collect-deny-compl* γ *rs* (*P* ∪ {*p*. *matches* γ *m Drop p*})) |
collect-deny-compl γ ((*Rule m Accept*)#*rs*) *P* = (*collect-deny-compl* γ *rs* (*P* ∪ {*p*. *matches* γ *m Accept p*}))

lemma *collect-deny-compl-correct*: *simple-ruleset rs* ⟹ (− *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* ∈ *P*. ¬ *matches* γ *r Accept p*} = − *P* ∪ {*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* ∈ *P*. ¬ *matches* γ *r Accept p*}] *set-simp1* **have**
$-$ *collect-deny-compl* γ *rs* ($-$ *P* ∪ *Collect* (*matches* γ *r Accept*)) = *collect-deny* γ *rs* {*p* ∈ *P*. ¬ *matches* γ *r Accept p*} **by** *simp*
thus *?case* **by** *auto*
next
case (*2* γ *r rs*)
have *set-simp1*: $-$ {*p* ∈ *P*. ¬ *matches* γ *r Drop p*} = $-$ *P* ∪ {*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*)

21.3 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 Undecided* = *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* ∈ *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* ∈ *packet-set-to-set ?*γ *?P*. ¬ *matches ?*γ *?m ?a p*}, it should be possible to build an executable version of the algorithm above.

21.4 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*

($packet\text{-}set\text{-}opt$ ($packet\text{-}set\text{-}constrain$ $Accept$ m P)) ($packet\text{-}set\text{-}opt$ ($collect\text{-}allow\text{-}impl\text{-}v2$ rs ($packet\text{-}set\text{-}opt$ ($packet\text{-}set\text{-}constrain\text{-}not$ $Accept$ m ($packet\text{-}set\text{-}opt$ P)))))) |
$collect\text{-}allow\text{-}impl\text{-}v2$ (($Rule$ m $Drop$)#rs) P = ($collect\text{-}allow\text{-}impl\text{-}v2$ rs ($packet\text{-}set\text{-}opt$ ($packet\text{-}set\text{-}constrain\text{-}not$ $Drop$ m ($packet\text{-}set\text{-}opt$ P))))

lemma *collect-allow-impl-v2*: *simple-ruleset rs* $\Longrightarrow$ *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* $\Longrightarrow$
(*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 *Analyze-TUM-Net-Firewall*
imports *Main ../../Output-Format/IPSpace-Format-Ln ../../Call-Return-Unfolding ../../Optimizing*
~~/src/HOL/Library/Code-Target-Nat
~~/src/HOL/Library/Code-Target-Int
~~/src/HOL/Library/Code-Char
../../Packet-Set
begin

22 Example: Chair for Network Architectures and Services (TUM)

definition *unfold-ruleset-FORWARD* :: *iptrule-match ruleset* ⇒ *iptrule-match rule list* **where**
unfold-ruleset-FORWARD rs = ((*optimize-matches opt-MatchAny-match-expr*) ^^*10*)

(*optimize-matches opt-simple-matcher* (*rw-Reject* (*rm-LogEmpty* (((*process-call rs*) ^^*5*) [*Rule MatchAny* (*Call ''FORWARD''*)])))))

definition *map-of-string* :: (*string* × *iptrule-match rule list*) *list* ⇒ *string* ⇀ *iptrule-match rule list* **where**
map-of-string rs = *map-of rs*

definition *upper-closure* :: *iptrule-match rule list* ⇒ *iptrule-match rule list* **where**
upper-closure rs == *rmMatchFalse* (((*optimize-matches opt-MatchAny-match-expr*) ^^*2000*) (*optimize-matches-a opt-simple-matcher-in-doubt-allow-extra rs*))
definition *lower-closure* :: *iptrule-match rule list* ⇒ *iptrule-match rule list* **where**
lower-closure rs == *rmMatchFalse* (((*optimize-matches opt-MatchAny-match-expr*) ^^*2000*) (*optimize-matches-a opt-simple-matcher-in-doubt-deny-extra rs*))

definition *deny-set* :: *iptrule-match rule list* ⇒ *iptrule-match packet-set list* **where**
deny-set rs ≡ *filter* (λ*a*. *a* ≠ *packet-set-UNIV*) (*map packet-set-opt* (*allow-set-not-inter rs*))

definition *bitmask-to-strange-inverse-cisco-mask*:: *nat* ⇒ (*nat* × *nat* × *nat* × *nat*) **where**
bitmask-to-strange-inverse-cisco-mask n ≡ *dotteddecimal-of-ipv4addr* ((*NOT* (((*mask*

n)::*ipv4addr*) << (*32* − *n*))))
lemma *bitmask-to-strange-inverse-cisco-mask 16* = (*0*, *0*, *255*, *255*) **by** *eval*
lemma *bitmask-to-strange-inverse-cisco-mask 24* = (*0*, *0*, *0*, *255*) **by** *eval*
lemma *bitmask-to-strange-inverse-cisco-mask 8* = (*0*, *255*, *255*, *255*) **by** *eval*
lemma *bitmask-to-strange-inverse-cisco-mask 32* = (*0*, *0*, *0*, *0*) **by** *eval*

export-code *unfold-ruleset-FORWARD map-of-string upper-closure lower-closure format-Ln-rules-uncompressed compress-Ln-ips does-I-has-compressed-rules*
Rule
Accept Drop Log Reject Call Return Empty Unknown
Match MatchNot MatchAnd MatchAny
Ip4Addr Ip4AddrNetmask
ProtAll ProtTCP ProtUDP
Src Dst Prot Extra
nat-of-integer integer-of-nat
UncompressedFormattedMatch Pos Neg
does-I-has-compressed-prots
bitmask-to-strange-inverse-cisco-mask
deny-set
in *SML* **module-name** *Test* **file** *unfold-code.ML*

ML-file *unfold-code.ML*

ML-file *iptables-Ln-29.11.2013.ML*

This is the firewall ruleset (excerpt, 24 of 2800 rules displayed) we are going to analyze:

```
Chain FORWARD (policy ACCEPT)
target   prot source             destination
LOG_DROP all  127.0.0.0/8        0.0.0.0/0
ACCEPT   tcp  131.159.14.206     0.0.0.0/0          multiport sports 389,636
ACCEPT   tcp  131.159.14.208     0.0.0.0/0          multiport sports 389,636
ACCEPT   udp  131.159.14.206     0.0.0.0/0          udp spt:88
ACCEPT   udp  131.159.14.208     0.0.0.0/0          udp spt:88
ACCEPT   tcp  131.159.14.192/27 0.0.0.0/0          tcp spt:3260
ACCEPT   tcp  131.159.14.0/23    131.159.14.192/27 tcp dpt:3260
ACCEPT   tcp  131.159.20.0/24    131.159.14.192/27 tcp dpt:3260
ACCEPT   udp  131.159.15.252     0.0.0.0/0
ACCEPT   udp  0.0.0.0/0          131.159.15.252     multiport
                                                        dports 4569,5000:65535
ACCEPT   all  131.159.15.247     0.0.0.0/0
ACCEPT   all  0.0.0.0/0          131.159.15.247
ACCEPT   all  131.159.15.248     0.0.0.0/0
ACCEPT   all  0.0.0.0/0          131.159.15.248
         tcp  0.0.0.0/0          131.159.14.0/23    state NEW tcp
               dpt:22flags: 0x17/0x02 recent: SET name: ratessh side: source
```

```
         tcp  0.0.0.0/0         131.159.20.0/23   state NEW tcp
               dpt:22flags: 0x17/0x02 recent: SET name: ratessh side: source
mac_96   all  131.159.14.0/25   0.0.0.0/0
LOG_DROP all !131.159.14.0/25   0.0.0.0/0

Chain LOG_DROP (21 references)
target   prot source            destination
LOG      all  0.0.0.0/0         0.0.0.0/0         limit: avg 100/min
                           burst 5  LOG flags 0 level 4 prefix "[IPT_DROP]:"
DROP     all  0.0.0.0/0         0.0.0.0/0

Chain mac_96 (1 references)
target   prot source            destination
RETURN   all  131.159.14.92     0.0.0.0/0         MAC XX:XX:XX:XX:XX:XX
DROP     all  131.159.14.92     0.0.0.0/0
RETURN   all  131.159.14.65     0.0.0.0/0         MAC XX:XX:XX:XX:XX:XX
DROP     all  131.159.14.65     0.0.0.0/0
```

ML⟪
open Test;
⟫
declare[[*ML-print-depth=50*]]
ML⟪
val rules = *unfold-ruleset-FORWARD* (*map-of-string firewall-chains*)
⟫
ML⟪
length rules;
val upper = *upper-closure rules*;
length upper;⟫
ML⟪
val lower = *lower-closure rules*;
length lower;⟫

How long does the unfolding take?

ML-val⟪
val t0 = *Time.now*();
val - = *unfold-ruleset-FORWARD* (*map-of-string firewall-chains*);
val t1= *Time.now*();
writeln(*String.concat* [*It took* , *Time.toString*(*Time.*−(*t1*,*t0*)), *seconds*])
⟫

on my system, less than 1 second.

Time required for calculating both closures

ML-val⟪
val t0 = *Time.now*();
val - = *upper-closure rules*;
val - = *lower-closure rules*;
val t1= *Time.now*();
writeln(*String.concat* [*It took* , *Time.toString*(*Time.*−(*t1*,*t0*)), *seconds*])
⟫

on my system, less than five seconds.

ML⟪

```
fun dump-dotteddecimal-ip (a,(b,(c,d))) = ^ Int.toString (integer-of-nat a) ^. ^ Int.toString
(integer-of-nat b) ^. ^ Int.toString (integer-of-nat c) ^. ^ Int.toString (integer-of-nat
d);

fun dump-ip (Ip4Addr ip) = (dump-dotteddecimal-ip ip) ^/32
 | dump-ip (Ip4AddrNetmask (ip, nm)) = (dump-dotteddecimal-ip ip) ^/ ^ Int.toString
(integer-of-nat nm);

fun dump-prot ProtAll = all
  | dump-prot ProtTCP = tcp
  | dump-prot ProtUDP = udp;

fun dump-prots [] = all
  | dump-prots [Pos p] = dump-prot p
  | dump-prots [Neg p] = ! ^dump-prot p;
  (*undefined otherwise*)

fun dump-extra [] = ;
   (*undefined case checks that we have no Extra (i.e. unknown in here)*)

fun dump-action Accept = ACCEPT
  | dump-action Drop = DROP
  | dump-action Log = LOG
  | dump-action Reject = REJECT
;

local
  fun dump-ip-list-hlp [] =
    | dump-ip-list-hlp ((Pos ip)::ips) = ((dump-ip ip) ^ dump-ip-list-hlp ips)
    | dump-ip-list-hlp ((Neg ip)::ips) = (! ^ (dump-ip ip) ^ dump-ip-list-hlp ips)
in
  fun dump-ip-list [] = 0.0.0.0/0
    | dump-ip-list rs = dump-ip-list-hlp rs
end;


fun dump-iptables [] = ()
  | dump-iptables ((UncompressedFormattedMatch (src, dst, proto, extra), a) :: rs)
=
      (writeln (dump-action a ^        ^
               ^ dump-prots proto ^   --   ^
               ^ dump-ip-list src ^              ^
               ^ dump-ip-list dst ^       ^
               ^ dump-extra extra); dump-iptables rs);

fun dump-iptables-save [] = ()
  | dump-iptables-save ((UncompressedFormattedMatch (src, dst, proto, []), a) ::
```

rs) =
  (*writeln* (−*A FORWARD* ˆ
    (*if List.length src* = *1 then* −*s* ˆ *dump-ip-list src* ˆ *else if List.length src* > *1 then ERROR else*) ˆ
    (*if List.length dst* = *1 then* −*d* ˆ *dump-ip-list dst* ˆ *else if List.length dst* > *1 then ERROR else*) ˆ
      (*if List.length proto* = *1 then* −*p* ˆ *dump-prots proto* ˆ *else if List.length proto* > *1 then ERROR else*) ˆ
     ˆ −*j* ˆ *dump-action a*); *dump-iptables-save rs*);
⟫

ML-val⟪
length (*format-Ln-rules-uncompressed upper*);
(*format-Ln-rules-uncompressed upper*);
⟫
— optimized upper closure, roughly 1k rules
ML-val⟪
length (*compress-Ln-ips* (*format-Ln-rules-uncompressed upper*));
⟫
ML-val⟪
(*compress-Ln-ips* (*format-Ln-rules-uncompressed upper*));
⟫
ML-val⟪
length (*does-I-has-compressed-rules* (*compress-Ln-ips* (*format-Ln-rules-uncompressed upper*)));
does-I-has-compressed-rules (*compress-Ln-ips* (*format-Ln-rules-uncompressed upper*));
⟫
ML-val⟪
does-I-has-compressed-prots (*compress-Ln-ips* (*format-Ln-rules-uncompressed upper*));
⟫

iptables -L -n

ML-val⟪
writeln Chain INPUT (*policy ACCEPT*);
writeln target prot opt source destination;
writeln ;
writeln Chain FORWARD (*policy ACCEPT*);
writeln target prot opt source destination;
dump-iptables (*compress-Ln-ips* (*format-Ln-rules-uncompressed upper*));

writeln Chain OUTPUT (*policy ACCEPT*);
writeln target prot opt source destination
⟫

Upper Closure (excerpt)

```
Chain FORWARD (policy ACCEPT)
```

```
target  prot source              destination
DROP    all  127.0.0.0/8         0.0.0.0/0
ACCEPT  tcp  131.159.14.206/32   0.0.0.0/0
ACCEPT  tcp  131.159.14.208/32   0.0.0.0/0
ACCEPT  udp  131.159.14.206/32   0.0.0.0/0
ACCEPT  udp  131.159.14.208/32   0.0.0.0/0
ACCEPT  tcp  131.159.14.192/27   0.0.0.0/0
ACCEPT  tcp  131.159.14.0/23     131.159.14.192/27
ACCEPT  tcp  131.159.20.0/24     131.159.14.192/27
ACCEPT  udp  131.159.15.252/32   0.0.0.0/0
ACCEPT  udp  0.0.0.0/0           131.159.15.252/32
ACCEPT  all  131.159.15.247/32   0.0.0.0/0
ACCEPT  all  0.0.0.0/0           131.159.15.247/32
ACCEPT  all  131.159.15.248/32   0.0.0.0/0
ACCEPT  all  0.0.0.0/0           131.159.15.248/32
DROP    all  !131.159.14.0/25    0.0.0.0/0
```

iptables -L -n

ML-val⟪
writeln Chain INPUT (*policy ACCEPT*);
writeln target   prot opt source   destination;
writeln ;
writeln Chain FORWARD (*policy ACCEPT*);
writeln target   prot opt source   destination;
dump-iptables (*compress-Ln-ips* (*format-Ln-rules-uncompressed lower*));

writeln Chain OUTPUT (*policy ACCEPT*);
writeln target   prot opt source   destination
⟫

Lower Closure (excerpt)

```
Chain FORWARD (policy ACCEPT)
target  prot source              destination
DROP    all  127.0.0.0/8         0.0.0.0/0
ACCEPT  udp  131.159.15.252/32   0.0.0.0/0
ACCEPT  all  131.159.15.247/32   0.0.0.0/0
ACCEPT  all  0.0.0.0/0           131.159.15.247/32
ACCEPT  all  131.159.15.248/32   0.0.0.0/0
ACCEPT  all  0.0.0.0/0           131.159.15.248/32
DROP    all  131.159.14.92/32    0.0.0.0/0
DROP    all  131.159.14.65/32    0.0.0.0/0
...  (unfolded DROPs from chain mac_96
DROP    all  !131.159.14.0/25    0.0.0.0/0
```

iptables-save

ML-val⟪
writeln # Generated by iptables−save v1.4.21 on Wed Sep 3 18:*02*:*01 2014*;
writeln ∗filter;
writeln :*INPUT ACCEPT* [*0*:*0*];

writeln :*FORWARD ACCEPT* [*0*:*0*];
writeln :*OUTPUT ACCEPT* [*0*:*0*];
dump-iptables-save (*compress-Ln-ips* (*format-Ln-rules-uncompressed upper*));
writeln COMMIT;
writeln # *Completed on Wed Sep 3 18*:*02*:*01 2014*;
⟫

Cisco

ML⟪
fun dump-action-cisco Accept = *permit*
 | *dump-action-cisco Drop* = *deny*
;

fun dump-prot-cisco [] = *ip*
 | *dump-prot-cisco* [*Pos ProtAll*] = *ip*
 | *dump-prot-cisco* [*Pos ProtTCP*] = *tcp*
 | *dump-prot-cisco* [*Pos ProtUDP*] = *udp*;

local
 fun dump-ip-cisco (*Ip4Addr ip*) = *host* ^(*dump-dotteddecimal-ip ip*)
 | *dump-ip-cisco* (*Ip4AddrNetmask* (*ip*, *nm*)) = (*dump-dotteddecimal-ip ip*)^ ^(*dump-dotteddecimal-ip* (*bitmask-to-strange-inverse-cisco-mask nm*));
in
 fun dump-ip-list-cisco [] = *any*
 | *dump-ip-list-cisco* [*Pos ip*] = *dump-ip-cisco ip*
 | *dump-ip-list-cisco* [*Neg ip*] = *TODO*^*dump-ip-cisco ip*
end;

fun dump-cisco [] = ()
 | *dump-cisco* ((*UncompressedFormattedMatch* (*src*, *dst*, *proto*, []), *a*) :: *rs*) =
 (*writeln* (*access*−*list 101* ^ *dump-action-cisco a* ^
 (*if List.length proto* <= *1 then* ^ *dump-prot-cisco proto* ^ *else ERROR*)^
 (*dump-ip-list-cisco src*)^ ^(*dump-ip-list-cisco dst*)); *dump-cisco rs*);
⟫

ML-val⟪
writeln interface fe0;
writeln ip address 10.*1*.*1*.*1 255*.*255*.*255*.*254*;
writeln ip access−*group 101 in*;
writeln !;
dump-cisco (*compress-Ln-ips* (*format-Ln-rules-uncompressed upper*));
(∗*access*−*list 101 deny ip host 10*.*1*.*1*.*2 any*
access−*list 101 permit tcp any host 192*.*168*.*5*.*10 eq 80*
access−*list 101 permit tcp any host 192*.*168*.*5*.*11 eq 25*
access−*list 101 deny any*∗)

```
writeln !;
writeln ! // need to give the end command;
writeln end;

⟫
```

ML⟪

```
fun dump-action-flowtable Accept = flood
  | dump-action-flowtable Drop = drop
;

local
  fun dump-ip-flowtable (Ip4Addr ip) = (dump-dotteddecimal-ip ip)
      | dump-ip-flowtable (Ip4AddrNetmask (ip, nm)) = (dump-dotteddecimal-ip
ip) ^/^ Int.toString (integer-of-nat nm);
in
  fun dump-ip-list-flowtable [] = *
    | dump-ip-list-flowtable [Pos ip] = dump-ip-flowtable ip
    | dump-ip-list-flowtable [Neg ip] = TODO^dump-ip-flowtable ip
end;

fun dump-flowtable [] = ()
  | dump-flowtable ((UncompressedFormattedMatch (src, dst, proto, []), a) :: rs) =
       (writeln ((if List.length proto <= 1 then  ^ dump-prot-cisco proto ^  else
ERROR)^
             nw-src=^(dump-ip-list-flowtable src)^ nw-dst=^(dump-ip-list-flowtable
dst) ^
               priority=^Int.toString (List.length rs)^
               action=^dump-action-flowtable a
              ); dump-flowtable rs);
⟫
```

ML-val⟪

```
(*ip nw-src=10.0.0.1/32 nw-dst=* priority=30000 action=flood*)

dump-flowtable (compress-Ln-ips (format-Ln-rules-uncompressed upper));
⟫
```

packet set (test)

ML⟪

```
val t0 = Time.now();
val deny-set-set = deny-set upper;
val t1= Time.now();
writeln(String.concat [It took , Time.toString(Time.-(t1,t0)),  seconds])
⟫
```

ML-val⟨⟨
length deny-set-set;
⟩⟩
ML-val⟨⟨
deny-set-set;
⟩⟩

end
theory *Analyze-SQRL-Shorewall*
imports *Main ../../Output-Format/IPSpace-Format-Ln ../../Call-Return-Unfolding ../../Optimizing*
~~/src/HOL/Library/Code-Target-Nat
~~/src/HOL/Library/Code-Target-Int
~~/src/HOL/Library/Code-Char
begin

23 Example: SQRL Shorewall

definition *unfold-ruleset-FORWARD* :: *iptrule-match ruleset* ⇒ *iptrule-match rule list* **where**
unfold-ruleset-FORWARD rs = ((*optimize-matches opt-MatchAny-match-expr*) ^^*10*)

(*optimize-matches opt-simple-matcher* (*rw-Reject* (*rm-LogEmpty* (((*process-call rs*) ^^*20*) [*Rule MatchAny* (*Call* ''*FORWARD*'')]))))

definition *unfold-ruleset-OUTPUT* :: *iptrule-match ruleset* ⇒ *iptrule-match rule list* **where**
unfold-ruleset-OUTPUT rs = ((*optimize-matches opt-MatchAny-match-expr*) ^^*10*)

(*optimize-matches opt-simple-matcher* (*rw-Reject* (*rm-LogEmpty* (((*process-call rs*) ^^*20*) [*Rule MatchAny* (*Call* ''*OUTPUT*'')]))))

definition *map-of-string* :: (*string* × *iptrule-match rule list*) *list* ⇒ *string* ⇀ *iptrule-match rule list* **where**
map-of-string rs = *map-of rs*

definition *upper-closure* :: *iptrule-match rule list* ⇒ *iptrule-match rule list* **where**
upper-closure rs == *rmMatchFalse* (((*optimize-matches opt-MatchAny-match-expr*) ^^*2000*) (*optimize-matches-a opt-simple-matcher-in-doubt-allow-extra rs*))
definition *lower-closure* :: *iptrule-match rule list* ⇒ *iptrule-match rule list* **where**
lower-closure rs == *rmMatchFalse* (((*optimize-matches opt-MatchAny-match-expr*) ^^*2000*) (*optimize-matches-a opt-simple-matcher-in-doubt-deny-extra rs*))

export-code *unfold-ruleset-OUTPUT map-of-string upper-closure lower-closure*

format-Ln-rules-uncompressed compress-Ln-ips does-I-has-compressed-rules
  Rule
  Accept Drop Log Reject Call Return Empty Unknown
  Match MatchNot MatchAnd MatchAny
  Ip4Addr Ip4AddrNetmask
  ProtAll ProtTCP ProtUDP
  Src Dst Prot Extra
  nat-of-integer integer-of-nat
  UncompressedFormattedMatch Pos Neg
  does-I-has-compressed-prots
  in *SML* **module-name** *Test* **file** *unfold-code.ML*

ML-file *unfold-code.ML*

ML-file *akachan−iptables−Ln.ML*

```
ML⟨
open Test;
⟩
declare[[ML-print-depth=50]]
ML⟨
val rules = unfold-ruleset-OUTPUT (map-of-string firewall-chains)
⟩
ML⟨
length rules;
val upper = upper-closure rules;
length upper;⟩
ML⟨
val lower = lower-closure rules;
length lower;⟩

ML⟨
fun dump-ip (Ip4Addr (a,(b,(c,d)))) = ^ Int.toString (integer-of-nat a) ^. ^ Int.toString
(integer-of-nat b) ^. ^ Int.toString (integer-of-nat c) ^. ^ Int.toString (integer-of-nat
d) ^/32
  | dump-ip (Ip4AddrNetmask ((a,(b,(c,d))), nm)) =
     ^ Int.toString (integer-of-nat a) ^. ^ Int.toString (integer-of-nat b) ^. ^ Int.toString
(integer-of-nat c) ^. ^ Int.toString (integer-of-nat d) ^/ ^ Int.toString (integer-of-nat
nm);

fun dump-prot ProtAll = all
  | dump-prot ProtTCP = tcp
  | dump-prot ProtUDP = udp;

fun dump-prots [] = all
  | dump-prots [Pos p] = dump-prot p
  | dump-prots [Neg p] = ! ^dump-prot p;
```

(*undefined otherwise*)

fun dump-extra [] = ;

fun dump-action Accept = *ACCEPT*
| *dump-action Drop* = *DROP*
| *dump-action Log* = *LOG*
| *dump-action Reject* = *REJECT*
;

local
fun dump-ip-list-hlp [] =
| *dump-ip-list-hlp* ((*Pos ip*)::*ips*) = ((*dump-ip ip*) ^ *dump-ip-list-hlp ips*)
| *dump-ip-list-hlp* ((*Neg ip*)::*ips*) = (! ^ (*dump-ip ip*) ^ *dump-ip-list-hlp ips*)
in
fun dump-ip-list [] = *0.0.0.0/0*
| *dump-ip-list rs* = *dump-ip-list-hlp rs*
end;

fun dump-iptables [] = ()
| *dump-iptables* ((*UncompressedFormattedMatch* (*src*, *dst*, *proto*, *extra*), *a*) :: *rs*)
=
(*writeln* (*dump-action a* ^ ^
^ *dump-prots proto* ^ −− ^
^ *dump-ip-list src* ^ ^
^ *dump-ip-list dst* ^ ^
^ *dump-extra extra*); *dump-iptables rs*);
⟫

ML-val⟪
length (*format-Ln-rules-uncompressed upper*);
(*format-Ln-rules-uncompressed upper*);
⟫
ML-val⟪
(*compress-Ln-ips* (*format-Ln-rules-uncompressed upper*));
⟫
ML-val⟪
length (*does-I-has-compressed-rules* (*compress-Ln-ips* (*format-Ln-rules-uncompressed upper*)));
does-I-has-compressed-rules (*compress-Ln-ips* (*format-Ln-rules-uncompressed upper*));
⟫
ML-val⟪
does-I-has-compressed-prots (*compress-Ln-ips* (*format-Ln-rules-uncompressed upper*));
⟫
ML-val⟪
dump-iptables (*compress-Ln-ips* (*format-Ln-rules-uncompressed upper*));

⟫

ML-val⟪
compress-Ln-ips (*format-Ln-rules-uncompressed lower*);
⟫
ML-val⟪
length (*does-I-has-compressed-rules* (*compress-Ln-ips* (*format-Ln-rules-uncompressed lower*)));
does-I-has-compressed-rules (*compress-Ln-ips* (*format-Ln-rules-uncompressed lower*));
⟫
ML-val⟪
does-I-has-compressed-prots (*compress-Ln-ips* (*format-Ln-rules-uncompressed lower*));
⟫

end
theory *Analyze-Synology-Diskstation*
imports *iptables-Ln-tuned-parsed../../Output-Format/IPSpace-Format-Ln ../../Call-Return-Unfolding ../../Optimizing*
../../Packet-Set
begin

24 Example: Synology Diskstation

we removed the established,related rule

definition *example-ruleset* == [*''DOS-PROTECT''* ↦ [*Rule* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Extra* (*''Prot icmp''*))) (*Match* (*Extra* (*''icmptype 8 limit*: *avg 1/sec burst 5''*)))))) (*Return*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Extra* (*''Prot icmp''*))) (*Match* (*Extra* (*''icmptype 8''*)))))) (*Drop*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Prot* (*ProtTCP*))) (*Match* (*Extra* (*''tcp flags*:*0x17/0x04 limit*: *avg 1/sec burst 5''*)))))) (*Return*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Prot* (*ProtTCP*))) (*Match* (*Extra* (*''tcp flags*:*0x17/0x04''*)))))) (*Drop*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Prot* (*ProtTCP*))) (*Match* (*Extra* (*''tcp flags*:*0x17/0x02 limit*: *avg 10000/sec burst 100''*)))))) (*Return*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Prot* (*ProtTCP*))) (*Match* (*Extra* (*''tcp flags*:*0x17/0x02''*)))))) (*Drop*)],
''INPUT'' ↦ [*Rule* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* ((*0*,*0*,*0*,*0*)) (*0*)))) (*MatchAnd* (*Match* (*Prot* (*ProtAll*))) (*MatchAny*)))) (*Call* (*''DOS-PROTECT''*)),

(∗ *Rule (MatchAnd (Match (Src (Ip4AddrNetmask ((0,0,0,0)) (0)))) (MatchAnd (Match (Dst (Ip4AddrNetmask ((0,0,0,0)) (0)))) (MatchAnd (Match (Prot (ProtAll))) (Match (Extra (''state RELATED,ESTABLISHED'')))))) (Accept),* ∗)
Rule (MatchAnd (Match (Src (Ip4AddrNetmask ((0,0,0,0)) (0)))) (MatchAnd (Match (Dst (Ip4AddrNetmask ((0,0,0,0)) (0)))) (MatchAnd (Match (Prot (ProtTCP))) (Match (Extra (''tcp dpt:22'')))))) (Drop),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask ((0,0,0,0)) (0)))) (MatchAnd (Match (Dst (Ip4AddrNetmask ((0,0,0,0)) (0)))) (MatchAnd (Match (Prot (ProtTCP))) (Match (Extra (''multiport dports 21,873,5005,5006,80,548,111,2049,892'')))))) (Drop),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask ((0,0,0,0)) (0)))) (MatchAnd (Match (Dst (Ip4AddrNetmask ((0,0,0,0)) (0)))) (MatchAnd (Match (Prot (ProtUDP))) (Match (Extra (''multiport dports 123,111,2049,892,5353'')))))) (Drop),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask ((192,168,0,0)) (16)))) (MatchAnd (Match (Dst (Ip4AddrNetmask ((0,0,0,0)) (0)))) (MatchAnd (Match (Prot (ProtAll))) (MatchAny)))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask ((0,0,0,0)) (0)))) (MatchAnd (Match (Dst (Ip4AddrNetmask ((0,0,0,0)) (0)))) (MatchAnd (Match (Prot (ProtAll))) (MatchAny)))) (Drop),
Rule (MatchAny) (Accept)],
''FORWARD'' ↦ *[Rule (MatchAny) (Accept)],*
''OUTPUT'' ↦ *[Rule (MatchAny) (Accept)]]*

abbreviation *MatchAndInfix* :: *'a match-expr* ⇒ *'a match-expr* ⇒ *'a match-expr*
(**infixr** *MATCHAND 65*) **where** *MatchAndInfix m1 m2* ≡ *MatchAnd m1 m2*

definition *example-ruleset-simplified* = ((*optimize-matches opt-MatchAny-match-expr*) ^^*10*)

(*optimize-matches opt-simple-matcher* (*rw-Reject* (*rm-LogEmpty* (((*process-call example-ruleset*) ^^*2*) [*Rule MatchAny* (*Call ''INPUT''*)]))))
value(*code*) *example-ruleset-simplified*

lemma *good-ruleset example-ruleset-simplified* **by** *eval*
lemma *simple-ruleset example-ruleset-simplified* **by** *eval*

packets from the local lan are allowed (in doubt)

value *approximating-bigstep-fun* (*simple-matcher*, *in-doubt-allow*) (|*src-ip*=*ipv4addr-of-dotteddecimal* (*192,168,3,5*), *dst-ip*=*0*, *prot*=*protPacket.ProtTCP*|)
example-ruleset-simplified
Undecided = *Decision FinalAllow*
lemma *approximating-bigstep-fun* (*simple-matcher*, *in-doubt-allow*) (|*src-ip*=*ipv4addr-of-dotteddecimal* (*192,168,3,5*), *dst-ip*=*0*, *prot*=*protPacket.ProtTCP*|)
example-ruleset-simplified
Undecided = *Decision FinalAllow* **by** *eval*

However, they might also be rate-limited, ... (we don't know about icmp)

lemma *approximating-bigstep-fun* (*simple-matcher*, *in-doubt-deny*) (|*src-ip*=*ipv4addr-of-dotteddecimal* (*192,168,3,5*), *dst-ip*=*0*, *prot*=*protPacket.ProtTCP*|)
example-ruleset-simplified

Undecided = *Decision FinalDeny* **by** *eval*

But we can guarantee that packets from the outside are blocked!

lemma *approximating-bigstep-fun* (*simple-matcher*, *in-doubt-allow*) (|*src-ip*=*ipv4addr-of-dotteddecimal* (*8*,*8*,*3*,*5*), *dst-ip*=*0*, *prot*=*protPacket.ProtTCP*|)
example-ruleset-simplified
Undecided = *Decision FinalDeny* **by** *eval*

lemma *wf-unknown-match-tac* $\alpha \Longrightarrow$ *approximating-bigstep-fun* (*simple-matcher*, α) *p* *example-ruleset-simplified* *s* = *approximating-bigstep-fun* (*simple-matcher*, α) *p* (((*process-call example-ruleset*) ^^*2*) [*Rule MatchAny* (*Call* ''*INPUT*'')]) *s*
apply(*simp add*: *example-ruleset-simplified-def*)
apply(*simp add*: *optimize-matches-opt-MatchAny-match-expr*)
apply(*simp add*: *opt-simple-matcher-correct*)
apply(*simp add*: *rw-Reject-fun-semantics*)
apply(*simp add*: *rm-LogEmpty-fun-semantics*)
done

in doubt allow closure

value *rmMatchFalse* (((*optimize-matches opt-MatchAny-match-expr*) ^^*10*) (*optimize-matches-a opt-simple-matcher-in-doubt-allow-extra example-ruleset-simplified*))

in doubt deny closure

value *rmMatchFalse* (((*optimize-matches opt-MatchAny-match-expr*) ^^*10*) (*optimize-matches-a opt-simple-matcher-in-doubt-deny-extra example-ruleset-simplified*))

upper closure

lemma *rmshadow* (*simple-matcher*, *in-doubt-allow*) (*rmMatchFalse* (((*optimize-matches opt-MatchAny-match-expr*) ^^*10*) (*optimize-matches-a opt-simple-matcher-in-doubt-allow-extra example-ruleset-simplified*))) *UNIV* =
[*Rule* (*Match* (*Src* (*Ip4AddrNetmask* (*192*, *168*, *0*, *0*) *16*))) *Accept*, *Rule MatchAny Drop*]
apply(*subst tmp*)
apply(*subst rmshadow.simps*)
apply(*simp del*: *rmshadow.simps*)
apply(*simp add*: *Matching-Ternary.matches-def*)
apply(*intro conjI impI*)
apply(*rule-tac x*=(|*src-ip*=*ipv4addr-of-dotteddecimal* (*8*,*8*,*3*,*5*), *dst-ip*=*0*, *prot*=*protPacket.ProtTCP*|) **in** *exI*)
apply(*simp add*: *ipv4addr-of-dotteddecimal.simps ipv4range-set-from-bitmask-def ipv4range-set-from-netmask-def Let-def ipv4addr-of-nat-def*)

apply(*thin-tac* $\exists$ *p*. *?x p*)
apply(*rule-tac x*=(|*src-ip*=*ipv4addr-of-dotteddecimal* (*192*,*168*,*99*,*0*), *dst-ip*=*0*, *prot*=*protPacket.ProtTCP*|) **in** *exI*)

apply(*simp add*: *ipv4addr-of-dotteddecimal.simps ipv4range-set-from-bitmask-def ipv4range-set-from-netmask-def Let-def ipv4addr-of-nat-def*)
done

lower closure

lemma *rmshadow* (*simple-matcher*, *in-doubt-deny*) (*rmMatchFalse* (((*optimize-matches opt-MatchAny-match-expr*) ^^*10*) (*optimize-matches-a opt-simple-matcher-in-doubt-deny-extra example-ruleset-simplified*))) *UNIV* =
  [*Rule MatchAny Drop*]
apply(*subst tmp'*)
apply(*subst rmshadow.simps*)
apply(*simp del*: *rmshadow.simps*)
apply(*simp add*: *Matching-Ternary.matches-def*)
done
hide-fact *tmp*

value *format-Ln-rules-uncompressed* [*Rule* (*Match* (*Src* (*Ip4AddrNetmask* (*192*, *168*, *0*, *0*) *16*))) *Accept*, *Rule MatchAny Drop*]

exact

value *format-Ln-rules-uncompressed example-ruleset-simplified*

value *length* (*example-ruleset-simplified*)

Wow, normalization has exponential?? blowup!!

value *length* (*normalize-rules example-ruleset-simplified*)
value *length* (*format-Ln-rules-uncompressed example-ruleset-simplified*)

thm *format-Ln-rules-uncompressed-correct*

upper closure

value *format-Ln-rules-uncompressed* (*rmMatchFalse* (((*optimize-matches opt-MatchAny-match-expr*) ^^*10*) (*optimize-matches-a opt-simple-matcher-in-doubt-allow-extra example-ruleset-simplified*)))
lemma *collect-allow-impl-v2* (*rmMatchFalse* (((*optimize-matches opt-MatchAny-match-expr*) ^^*10*) (*optimize-matches-a opt-simple-matcher-in-doubt-allow-extra example-ruleset-simplified*))) *packet-set-UNIV* =
  PacketSet [[(*Pos* (*Src* (*Ip4AddrNetmask* (*192*, *168*, *0*, *0*) *16*)), *Pos Accept*)]] **by** *eval*

lower closure

value *format-Ln-rules-uncompressed* (*rmMatchFalse* (((*optimize-matches opt-MatchAny-match-expr*) ^^*10*) (*optimize-matches-a opt-simple-matcher-in-doubt-deny-extra example-ruleset-simplified*)))
lemma *collect-allow-impl-v2* (*rmMatchFalse* (((*optimize-matches opt-MatchAny-match-expr*) ^^*10*) (*optimize-matches-a opt-simple-matcher-in-doubt-deny-extra example-ruleset-simplified*))) *packet-set-UNIV* =
  packet-set-Empty **by** *eval*

packet set test

value(*code*) *collect-allow-impl-v2* (*take 1 example-ruleset-simplified*) *packet-set-UNIV*

value(*code*) *collect-allow-impl-v2* (*take 2 example-ruleset-simplified*) *packet-set-UNIV*

end
theory *Analyze-Ringofsaturn-com*
imports
../../Call-Return-Unfolding
../../Optimizing
../../Output-Format/IPSpace-Format-Ln
../../Packet-Set
begin

25 Example: ringofsaturn.com

We have directly executable approximating semantics: *wf-ruleset* $?\gamma$ $?p$ $?rs$ $\Longrightarrow$ $?\gamma,?p\vdash \langle ?rs,\ ?s\rangle \Rightarrow_\alpha ?t$ = (*approximating-bigstep-fun* $?\gamma$ $?p$ $?rs$ $?s$ = $?t$)

value(*code*) *approximating-bigstep-fun* (*simple-matcher*, *in-doubt-allow*) (|*src-ip*=*0*, *dst-ip*=*0*, *prot*=*protPacket.ProtTCP*|)
(*process-call* [''*FORWARD*'' ↦ [*Rule* (*Match* (*Src* (*Ip4Addr*(*192*,*168*,*0*,*0*)))) *Drop*, *Rule MatchAny Accept*], ''*foo*'' ↦ []] [*Rule MatchAny* (*Call* ''*FORWARD*'')])
Undecided

definition *example-ruleset* == [''*FORWARD*'' ↦ [*Rule* (*Match* (*Src* ((*Ip4AddrNetmask* (*192*,*168*,*0*,*0*) *16*)))) (*Call* ''*foo*''), *Rule MatchAny Drop*],
''*foo*'' ↦ [*Rule MatchAny Log*, *Rule* (*Match* (*Extra* ''*foobar*'')) *Accept*]]

definition *example-ruleset-simplified* = *rm-LogEmpty* (((*process-call example-ruleset*) ^^*2*) [*Rule MatchAny* (*Call* ''*FORWARD*'')])
value *example-ruleset-simplified*

value *good-ruleset example-ruleset-simplified*
value *simple-ruleset example-ruleset-simplified*

lemma *approximating-bigstep-fun* (*simple-matcher*, *in-doubt-allow*) (|*src-ip*=*ipv4addr-of-dotteddecimal* (*192*,*168*,*3*,*5*), *dst-ip*=*0*, *prot*=*protPacket.ProtTCP*|)
example-ruleset-simplified
Undecided = *Decision FinalAllow* **by** *eval*
hide-const *example-ruleset-simplified example-ruleset*

25.1 Example Ruleset 1

definition *example-firewall* ≡ [''STATEFUL'' ↦ [Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll)) (Match (Extra ''state RELATED,ESTABLISHED''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll)) (Match (Extra ''state NEW''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll)) (MatchAny)))) (Call ''DUMP'')] ,
''DUMP'' ↦ [Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP)) (Match (Extra ''LOG flags 0 level 4''))))) (Log),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtUDP)) (Match (Extra ''LOG flags 0 level 4''))))) (Log),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP)) (Match (Extra ''reject−with tcp−reset''))))) (Reject),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtUDP)) (Match (Extra ''reject−with icmp−port−unreachable''))))) (Reject),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll)) (MatchAny)))) (Drop)] ,
''INPUT'' ↦ [Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll)) (MatchAny)))) (Call ''STATEFUL''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll)) (MatchAny)))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 8))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll)) (MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (10,0,0,0) 8))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll)) (MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (127,0,0,0) 8))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll)) (MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (169,254,0,0) 16))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll)) (MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (172,16,0,0) 12))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll)) (MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (224,0,0,0) 3))) (MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))

(MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (240,0,0,0) 8))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (160,86,0,0) 16))) (MatchAnd
(Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Drop),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Extra ''Prot icmp''))
(Match (Extra ''icmptype 3''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Extra ''Prot icmp''))
(Match (Extra ''icmptype 11''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Extra ''Prot icmp''))
(Match (Extra ''icmptype 0''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Extra ''Prot icmp''))
(Match (Extra ''icmptype 8''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:111''))))) (Drop),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:113 reject−with tcp−reset''))))) (Reject),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:4''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:20''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:21''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtUDP))
(Match (Extra ''udp dpt:20''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtUDP))
(Match (Extra ''udp dpt:21''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:22''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtUDP))
(Match (Extra ''udp dpt:22''))))) (Accept),

Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot* *ipt-protocol.ProtTCP*)) (*Match* (*Extra* ''*tcp dpt*:*80*''))))) (*Accept*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot* *ipt-protocol.ProtUDP*)) (*Match* (*Extra* ''*udp dpt*:*80*''))))) (*Accept*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot* *ipt-protocol.ProtTCP*)) (*Match* (*Extra* ''*tcp dpt*:*443*''))))) (*Accept*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot* *ipt-protocol.ProtUDP*)) (*Match* (*Extra* ''*udp dpt*:*443*''))))) (*Accept*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot* *ipt-protocol.ProtUDP*)) (*Match* (*Extra* ''*udp dpt*:*520 reject*−*with icmp*−*port*−*unreachable*''))))) (*Reject*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot* *ipt-protocol.ProtTCP*)) (*Match* (*Extra* ''*tcp dpts*:*137*:*139 reject*−*with icmp*−*port*−*unreachable*''))))) (*Reject*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot* *ipt-protocol.ProtUDP*)) (*Match* (*Extra* ''*udp dpts*:*137*:*139 reject*−*with icmp*−*port*−*unreachable*''))))) (*Reject*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot* *ipt-protocol.ProtAll*)) (*MatchAny*)))) (*Call* ''*DUMP*''),
Rule MatchAny (*Accept*)] ,
''*FORWARD*'' ↦ [*Rule MatchAny* (*Accept*)] ,
''*OUTPUT*'' ↦ [*Rule* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot* *ipt-protocol.ProtAll*)) (*MatchAny*)))) (*Accept*),
Rule MatchAny (*Accept*)]]

definition *simple-example-firewall* ≡ (((*optimize-matches opt-MatchAny-match-expr*) ^^*10*) (*optimize-matches opt-simple-matcher* (*rw-Reject* (*rm-LogEmpty* (((*process-call example-firewall*) ^^*3*) [*Rule MatchAny* (*Call* ''*INPUT*'')]))))))

It accepts everything in state RELATED,ESTABLISHED,NEW

value(*code*) *simple-example-firewall*
value *good-ruleset simple-example-firewall*
value *simple-ruleset simple-example-firewall*

lemma *approximating-bigstep-fun* (*simple-matcher*, *in-doubt-allow*) *p simple-example-firewall s* = *approximating-bigstep-fun* (*simple-matcher*, *in-doubt-allow*) *p* (((*process-call example-firewall*) ^^*3*) [*Rule MatchAny* (*Call* ''*INPUT*'')]) *s*
apply(*simp add*: *simple-example-firewall-def*)
apply(*simp add*: *optimize-matches-opt-MatchAny-match-expr*)
apply(*simp add*: *opt-simple-matcher-correct*)
apply(*simp add*: *rw-Reject-fun-semantics wf-in-doubt-allow*)
apply(*simp add*: *rm-LogEmpty-fun-semantics*)

done

value(*code*) ((*optimize-matches opt-MatchAny-match-expr*) ^^*10*) (*optimize-matches-a opt-simple-matcher-in-doubt-allow-extra simple-example-firewall*)

lemma *rmshadow* (*simple-matcher*, *in-doubt-allow*) (((*optimize-matches opt-MatchAny-match-expr*) ^^*10*) (*optimize-matches-a opt-simple-matcher-in-doubt-allow-extra simple-example-firewall*))
UNIV =
[*Rule MatchAny Accept*]
apply(*subst tmp*)
apply(*subst rmshadow.simps*)
apply(*simp del*: *rmshadow.simps*)
apply(*simp add*: *Matching-Ternary.matches-def*)
done

value(*code*) *approximating-bigstep-fun* (*simple-matcher*, *in-doubt-allow*) (|*src-ip*=*0*, *dst-ip*=*0*, *prot*=*protPacket.ProtTCP*|)
simple-example-firewall
Undecided
value(*code*) *approximating-bigstep-fun* (*simple-matcher*, *in-doubt-allow*) (|*src-ip*=*ipv4addr-of-dotteddecimal* (*192*,*168*,*3*,*5*), *dst-ip*=*0*, *prot*=*protPacket.ProtTCP*|)
simple-example-firewall
Undecided

We removed the first matches on state

definition *example-firewall2* ≡ [''*STATEFUL*'' ↦ [*Rule* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot ipt-protocol.ProtAll*)) (*Match* (*Extra* ''*state RELATED*,*ESTABLISHED*''))))) (*Accept*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot ipt-protocol.ProtAll*)) (*Match* (*Extra* ''*state NEW*''))))) (*Accept*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot ipt-protocol.ProtAll*)) (*MatchAny*)))) (*Call* ''*DUMP*'')] ,
''*DUMP*'' ↦ [*Rule* (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot ipt-protocol.ProtTCP*)) (*Match* (*Extra* ''*LOG flags 0 level 4*''))))) (*Log*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot ipt-protocol.ProtUDP*)) (*Match* (*Extra* ''*LOG flags 0 level 4*''))))) (*Log*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Dst* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match* (*Prot ipt-protocol.ProtTCP*)) (*Match* (*Extra* ''*reject−with tcp−reset*''))))) (*Reject*),
Rule (*MatchAnd* (*Match* (*Src* (*Ip4AddrNetmask* (*0*,*0*,*0*,*0*) *0*))) (*MatchAnd* (*Match*

(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtUDP))
(Match (Extra ''reject−with icmp−port−unreachable''))))) (Reject),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Drop)] ,
''INPUT'' ↦ [
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 8))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (10,0,0,0) 8))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (127,0,0,0) 8))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (169,254,0,0) 16))) (MatchAnd
(Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (172,16,0,0) 12))) (MatchAnd
(Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (224,0,0,0) 3))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (240,0,0,0) 8))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Call ''DUMP''),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (160,86,0,0) 16))) (MatchAnd
(Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Drop),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Extra ''Prot icmp''))
(Match (Extra ''icmptype 3''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Extra ''Prot icmp''))
(Match (Extra ''icmptype 11''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Extra ''Prot icmp''))
(Match (Extra ''icmptype 0''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Extra ''Prot icmp''))
(Match (Extra ''icmptype 8''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:111''))))) (Drop),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match

(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:113 reject−with tcp−reset''))))) (Reject),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:4''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:20''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:21''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtUDP))
(Match (Extra ''udp dpt:20''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtUDP))
(Match (Extra ''udp dpt:21''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:22''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtUDP))
(Match (Extra ''udp dpt:22''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:80''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtUDP))
(Match (Extra ''udp dpt:80''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpt:443''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtUDP))
(Match (Extra ''udp dpt:443''))))) (Accept),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtUDP))
(Match (Extra ''udp dpt:520 reject−with icmp−port−unreachable''))))) (Reject),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtTCP))
(Match (Extra ''tcp dpts:137:139 reject−with icmp−port−unreachable''))))) (Reject),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtUDP))
(Match (Extra ''udp dpts:137:139 reject−with icmp−port−unreachable''))))) (Reject),
Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match
(Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot ipt-protocol.ProtAll))
(MatchAny)))) (Call ''DUMP''),
Rule MatchAny (Accept)] ,
''FORWARD'' ↦ [Rule MatchAny (Accept)] ,

''OUTPUT'' ↦ [Rule (MatchAnd (Match (Src (Ip4AddrNetmask (0,0,0,0) 0)))
(MatchAnd (Match (Dst (Ip4AddrNetmask (0,0,0,0) 0))) (MatchAnd (Match (Prot
ipt-protocol.ProtAll)) (MatchAny)))) (Accept),
Rule MatchAny (Accept)]]

definition simple-example-firewall2 ≡ (((optimize-matches opt-MatchAny-match-expr) ^^10)
(optimize-matches opt-simple-matcher (rw-Reject (rm-LogEmpty (((process-call example-firewall2) ^^3)
[Rule MatchAny (Call ''INPUT'')])))))

lemma wf-unknown-match-tac α ⟹ approximating-bigstep-fun (simple-matcher,
α) p simple-example-firewall2 s = approximating-bigstep-fun (simple-matcher, α)
p (((process-call example-firewall2) ^^3) [Rule MatchAny (Call ''INPUT'')]) s
apply(simp add: simple-example-firewall2-def)
apply(simp add: optimize-matches-opt-MatchAny-match-expr)
apply(simp add: opt-simple-matcher-correct)
apply(simp add: rw-Reject-fun-semantics)
apply(simp add: rm-LogEmpty-fun-semantics)
done

value(code) simple-example-firewall2
value(code) zip (upto 0 (int (length simple-example-firewall2))) simple-example-firewall2
value good-ruleset simple-example-firewall2
value simple-ruleset simple-example-firewall2

in doubt allow closure

value(code) rmMatchFalse (((optimize-matches opt-MatchAny-match-expr) ^^10)
(optimize-matches-a opt-simple-matcher-in-doubt-allow-extra simple-example-firewall2))

in doubt deny closure

value(code) rmMatchFalse (((optimize-matches opt-MatchAny-match-expr) ^^10)
(optimize-matches-a opt-simple-matcher-in-doubt-deny-extra simple-example-firewall2))

value(code) format-Ln-rules-uncompressed (rmMatchFalse (((optimize-matches opt-MatchAny-match-expr) ^^
(optimize-matches-a opt-simple-matcher-in-doubt-allow-extra simple-example-firewall2)))
value(code) format-Ln-rules-uncompressed (rmMatchFalse (((optimize-matches opt-MatchAny-match-expr) ^^
(optimize-matches-a opt-simple-matcher-in-doubt-deny-extra simple-example-firewall2)))
value(code) format-Ln-rules-uncompressed simple-example-firewall2

Allowed Packets

the first 10 rules basically accept no packets

lemma collect-allow-impl-v2 (take 10 simple-example-firewall2) packet-set-UNIV
= packet-set-Empty **by** eval

value(code) allow-set-not-inter simple-example-firewall2

value(*code*) *map packet-set-opt* (*allow-set-not-inter simple-example-firewall2*)

end