Routing

Julius Michaelis, Cornelius Diekmann

March 17, 2025

Abstract

This entry contains definitions for routing with routing tables/longest prefix matching.

A routing table entry is modelled as a record of a prefix match, a metric, an output port, and an optional next hop. A routing table is a list of entries, sorted by prefix length and metric. Additionally, a parser and serializer for the output of the ip-route command, a function to create a relation from output port to corresponding destination IP space, and a model of a linux style router are included.

Contents

Sorting a list by two keys

theory *Linorder-Helper*
imports *Main*
begin

Sorting is fun...

The problem is that Isabelle does not have anything like `sortBy`, only *sort-key*. This means that there is no way to sort something based on two properties, with one being infinitely more important.

Enter this:

datatype $('a,'b)$ *linord-helper* = *LinordHelper* $'a$ $'b$

instantiation *linord-helper* :: (*linorder*, *linorder*) *linorder*
begin
 definition *linord-helper-less-eq1* a b $\equiv$ (*case* a *of* *LinordHelper* *a1* *a2* $\Rightarrow$ *case* b *of* *LinordHelper* *b1* *b2* $\Rightarrow$ *a1* $<$ *b1* $\vee$ *a1* $=$ *b1* $\wedge$ *a2* $\leq$ *b2*)
 definition $a \leq b \longleftrightarrow$ *linord-helper-less-eq1* a b
 definition $a < b \longleftrightarrow (a \neq b \wedge$ *linord-helper-less-eq1* a $b)$
instance
by *standard* (*auto simp*: *linord-helper-less-eq1-def less-eq-linord-helper-def less-linord-helper-def split*: *linord-helper.splits*)
end
lemmas *linord-helper-less* = *less-linord-helper-def linord-helper-less-eq1-def*
lemmas *linord-helper-le* = *less-eq-linord-helper-def linord-helper-less-eq1-def*

Now, it is possible to use *sort-key* f, with f constructing a *LinordHelper* containing the two desired properties for sorting.

end

1 Routing Table

theory *Routing-Table*
imports *IP-Addresses.Prefix-Match*
  IP-Addresses.IPv4 *IP-Addresses.IPv6*
  Linorder-Helper
  IP-Addresses.Prefix-Match-toString
  Pure−ex.Guess
begin

This section makes the necessary definitions to work with a routing table using longest prefix matching.

1.1 Definition

record(**overloaded**) *'i routing-action* =
 output-iface :: *string*
 next-hop :: *'i word option*

record(**overloaded**) *'i routing-rule* =
 routing-match :: (*'i*::*len*) *prefix-match*
 metric :: *nat*
 routing-action :: *'i routing-action*

This definition is engineered to model routing tables on packet forwarding devices. It eludes, e.g., the source address hint, which is only relevant for packets originating from the device itself.

context
begin

definition *default-metric = 0*

type-synonym *′i prefix-routing* = (*′i routing-rule*) *list*

abbreviation *routing-oiface a* ≡ *output-iface* (*routing-action a*)
abbreviation *routing-prefix r* ≡ *pfxm-length* (*routing-match r*)

definition *valid-prefixes* **where**
  valid-prefixes r = foldr conj (*map* (λ*rr. valid-prefix* (*routing-match rr*)) *r*) *True*
lemma *valid-prefixes-split*: *valid-prefixes* (*r#rs*) ⟹ *valid-prefix* (*routing-match r*) ∧ *valid-prefixes rs*
  using *valid-prefixes-def* **by** *force*

lemma *foldr-True-set*: *foldr* (λ*x.* (∧) (*f x*)) *l True* = (∀ *x* ∈ *set l. f x*)
  by (*induction l*) *simp-all*
lemma *valid-prefixes-alt-def*: *valid-prefixes r* = (∀ *e* ∈ *set r. valid-prefix* (*routing-match e*))
  unfolding *valid-prefixes-def*
  unfolding *foldr-map*
  unfolding *comp-def*
  unfolding *foldr-True-set*
  ..

fun *has-default-route* :: (*′i*::*len*) *prefix-routing* ⇒ *bool* **where**
has-default-route (*r#rs*) = (((*pfxm-length* (*routing-match r*)) = *0*) ∨ *has-default-route rs*) |
has-default-route Nil = False

lemma *has-default-route-alt*: *has-default-route rt* ⟷ (∃ *r* ∈ *set rt. pfxm-length* (*routing-match r*) = *0*) **by**(*induction rt*) *simp-all*

1.2 Single Packet Semantics

fun *routing-table-semantics* :: (*′i*::*len*) *prefix-routing* ⇒ *′i word* ⇒ *′i routing-action*
where
routing-table-semantics [] - = *routing-action* (*undefined*::*′i routing-rule*) |
routing-table-semantics (*r#rs*) *p* = (*if prefix-match-semantics* (*routing-match r*) *p then routing-action r else routing-table-semantics rs p*)
lemma *routing-table-semantics-ports-from-table*: *valid-prefixes rtbl* ⟹ *has-default-route rtbl* ⟹
  routing-table-semantics rtbl packet = r ⟹ *r* ∈ *routing-action* ‘ *set rtbl*
proof(*induction rtbl*)
  case (*Cons r rs*)

note *v-pfxs* = *valid-prefixes-split*[*OF Cons.prems*(*1*)]
show *?case*
proof(*cases pfxm-length* (*routing-match r*) = *0*)
case *True*
note *zero-prefix-match-all*[*OF conjunct1*[*OF v-pfxs*] *True*] *Cons.prems*(*3*)
then show *?thesis* **by** *simp*
next
case *False*
hence *has-default-route rs* **using** *Cons.prems*(*2*) **by** *simp*
from *Cons.IH*[*OF conjunct2*[*OF v-pfxs*] *this*] *Cons.prems*(*3*) **show** *?thesis* **by** *force*
qed
qed *simp*

1.3 Longest Prefix Match

We can abuse *LinordHelper* to sort.

definition *routing-rule-sort-key* ≡ *λr. LinordHelper* (*0* − (*of-nat* :: *nat* ⇒ *int*) (*pfxm-length* (*routing-match r*))) (*metric r*)

There is actually a slight design choice here. We can choose to sort based on (*?a* ≤ *?b*) = (*if pfxm-length ?a* = *pfxm-length ?b then pfxm-prefix ?a* ≤ *pfxm-prefix ?b else pfxm-length ?b* < *pfxm-length ?a*) (thus including the address) or only the prefix length (excluding it). Which is taken does not matter gravely, since the bits of the prefix can't matter. They're either eqal or the rules don't overlap and the metric decides. (It does matter for the resulting list though.) Ignoring the prefix and taking only its length is slightly easier.

definition *rr-ctor m l a nh me* ≡ ⦇ *routing-match* = *PrefixMatch* (*ipv4addr-of-dotdecimal m*) *l*, *metric* = *me*, *routing-action* =⦇*output-iface* = *a*, *next-hop* = (*map-option ipv4addr-of-dotdecimal nh*)⦈ ⦈
value *sort-key routing-rule-sort-key* [
rr-ctor (*0*,*0*,*0*,*1*) *3* '''' *None 0*,
rr-ctor (*0*,*0*,*0*,*2*) *8* [] *None 0*,
rr-ctor (*0*,*0*,*0*,*3*) *4* [] *None 13*,
rr-ctor (*0*,*0*,*0*,*3*) *4* [] *None 42*]

definition *is-longest-prefix-routing* ≡ *sorted* ∘ *map routing-rule-sort-key*

definition *correct-routing* :: (*'i*::*len*) *prefix-routing* ⇒ *bool* **where**
correct-routing r ≡ *is-longest-prefix-routing r* ∧ *valid-prefixes r*

Many proofs and functions around routing require at least parts of *correct-routing* as an assumption. Obviously, *correct-routing* is not given for arbitrary routing tables. Therefore, *correct-routing* is made to be executable and should be checked for any routing table after parsing. Note: *correct-routing* used to also require *has-default-route*, but none of the proofs require it anymore and it is not given for any routing table.

lemma *is-longest-prefix-routing-rule-exclusion*:
  assumes *is-longest-prefix-routing* (*r1* # *rn* # *rss*)
  shows *is-longest-prefix-routing* (*r1* # *rss*)
using *assms* **by**(*case-tac rss*) (*auto simp add*: *is-longest-prefix-routing-def*)

lemma *int-of-nat-less*: *int-of-nat a* < *int-of-nat b* ⟹ *a* < *b* **by** (*simp add*: *int-of-nat-def*)

lemma *is-longest-prefix-routing-sorted-by-length*:
  assumes *is-longest-prefix-routing r*
    and *r* = *r1* # *rs* @ *r2* # *rss*
  shows (*pfxm-length* (*routing-match r1*) ≥ *pfxm-length* (*routing-match r2*))
using *assms*
proof(*induction rs arbitrary*: *r*)
  case (*Cons rn rs*)
  let *?ro* = *r1* # *rs* @ *r2* # *rss*
  have *is-longest-prefix-routing ?ro* **using** *Cons.prems is-longest-prefix-routing-rule-exclusion*[*of r1 rn rs* @ *r2* # *rss*] **by** *simp*
  from *Cons.IH*[*OF this*] **show** *?case* **by** *simp*
next
  case *Nil* **thus** *?case* **by**(*auto simp add*: *is-longest-prefix-routing-def routing-rule-sort-key-def linord-helper-less-eq1-def less-eq-linord-helper-def int-of-nat-def*)
qed

definition *sort-rtbl* :: (′*i*::*len*) *routing-rule list* ⇒ ′*i routing-rule list* ≡ *sort-key routing-rule-sort-key*

lemma *is-longest-prefix-routing-sort*: *is-longest-prefix-routing* (*sort-rtbl r*) **unfolding** *sort-rtbl-def is-longest-prefix-routing-def* **by** *simp*

definition *unambiguous-routing rtbl* ≡ (∀ *rt1 rt2 rr ra*. *rtbl* = *rt1* @ *rr* # *rt2* ⟶ *ra* ∈ *set* (*rt1* @ *rt2*) ⟶ *routing-match rr* = *routing-match ra* ⟶ *routing-rule-sort-key rr* ≠ *routing-rule-sort-key ra*)
lemma *unambiguous-routing-Cons*: *unambiguous-routing* (*r* # *rtbl*) ⟹ *unambiguous-routing rtbl*
  unfolding *unambiguous-routing-def* **by**(*clarsimp*) (*metis append-Cons in-set-conv-decomp*)
lemma *unambiguous-routing* (*rr* # *rtbl*) ⟹ *is-longest-prefix-routing* (*rr* # *rtbl*) ⟹ *ra* ∈ *set rtbl* ⟹ *routing-match rr* = *routing-match ra* ⟹ *routing-rule-sort-key rr* < *routing-rule-sort-key ra*
  unfolding *is-longest-prefix-routing-def unambiguous-routing-def* **by**(*fastforce*)
primrec *unambiguous-routing-code* **where**
unambiguous-routing-code [] = *True* |
unambiguous-routing-code (*rr*#*rtbl*) = (*list-all* (λ*ra*. *routing-match rr* ≠ *routing-match ra* ∨ *routing-rule-sort-key rr* ≠ *routing-rule-sort-key ra*) *rtbl* ∧ *unambiguous-routing-code rtbl*)
lemma *unambiguous-routing-code*[*code-unfold*]: *unambiguous-routing rtbl* ⟷ *unambiguous-routing-code rtbl*
proof(*induction rtbl*)
  case (*Cons rr rtbl*) **show** *?case* (**is** *?l* ⟷ *?r*) **proof**
    assume *l*: *?l*

with *unambiguous-routing-Cons Cons.IH* **have** *unambiguous-routing-code rtbl* **by** *blast*
moreover have *list-all* (λ*ra. routing-match rr* ≠ *routing-match ra* ∨ *routing-rule-sort-key rr* ≠ *routing-rule-sort-key ra*) *rtbl*
using *l* **unfolding** *unambiguous-routing-def* **by**(*fastforce simp add*: *list-all-iff*)
ultimately show *?r* **by** *simp*
next
assume *r*: *?r*
with *Cons.IH* **have** *unambiguous-routing rtbl* **by** *simp*
from *r* **have** ∗: *list-all* (λ*ra. routing-match rr* ≠ *routing-match ra* ∨ *routing-rule-sort-key rr* ≠ *routing-rule-sort-key ra*) *rtbl* **by** *simp*
have *False* **if** *rr* # *rtbl* = *rt1* @ *rra* # *rt2 ra* ∈ *set* (*rt1* @ *rt2*) *routing-rule-sort-key rra* = *routing-rule-sort-key ra* ∧ *routing-match rra* = *routing-match ra* **for** *rt1 rt2 rra ra*
proof(*cases rt1* = [])
case *True* **thus** *?thesis* **using** *that* ∗ **by**(*fastforce simp add*: *list-all-iff*)
next
case *False*
with *that*(*1*) **have** *rtbl*: *rtbl* = *tl rt1* @ *rra* # *rt2* **by** (*metis list.sel*(*3*) *tl-append2*)
show *?thesis* **proof**(*cases ra* = *hd rt1*)
case *False* **hence** *ra* ∈ *set* (*tl rt1* @ *rt2*) **using** *that* **by**(*cases rt1*; *simp*)
with ‹*unambiguous-routing rtbl*› **show** *?thesis* **using** *that*(*3*) *rtbl* **unfolding** *unambiguous-routing-def* **by** *fast*
next
case *True* **hence** *rr* = *ra* **using** *that* ‹*rt1* ≠ []› **by**(*cases rt1*; *simp*)
thus *?thesis* **using** *that* ∗ **unfolding** *rtbl* **by**(*fastforce simp add*: *list-all-iff*)
qed
qed
thus *?l* **unfolding** *unambiguous-routing-def* **by** *blast*
qed
qed(*simp add*: *unambiguous-routing-def*)

lemma *unambigous-prefix-routing-weak-mono*:
assumes *lpfx*: *is-longest-prefix-routing* (*rr*#*rtbl*)
assumes *e*:*rr′* ∈ *set rtbl*
shows *routing-rule-sort-key rr′* ≥ *routing-rule-sort-key rr*
using *assms* **by**(*simp add*: *is-longest-prefix-routing-def*)
lemma *unambigous-prefix-routing-strong-mono*:
assumes *lpfx*: *is-longest-prefix-routing* (*rr*#*rtbl*)
assumes *uam*: *unambiguous-routing* (*rr*#*rtbl*)
assumes *e*:*rr′* ∈ *set rtbl*
assumes *ne*: *routing-match rr′* = *routing-match rr*
shows *routing-rule-sort-key rr′* > *routing-rule-sort-key rr*
proof −
from *uam e ne* **have** *routing-rule-sort-key rr* ≠ *routing-rule-sort-key rr′* **by**(*fastforce simp add*: *unambiguous-routing-def*)
moreover from *unambigous-prefix-routing-weak-mono lpfx e* **have** *routing-rule-sort-key rr* ≤ *routing-rule-sort-key rr′* **.**

ultimately show *?thesis* **by** *simp*
qed

lemma *routing-rule-sort-key* (*rr-ctor* (*0,0,0,0*) *8* [] *None 0*) > *routing-rule-sort-key* (*rr-ctor* (*0,0,0,0*) *24* [] *None 0*) **by** *eval*

In case you don't like that formulation of *is-longest-prefix-routing* over sorting, this is your lemma.

theorem *existential-routing*: *valid-prefixes rtbl* ⟹ *is-longest-prefix-routing rtbl* ⟹ *has-default-route rtbl* ⟹ *unambiguous-routing rtbl* ⟹
routing-table-semantics rtbl addr = *act* ⟷ (∃ *rr* ∈ *set rtbl*. *prefix-match-semantics* (*routing-match rr*) *addr* ∧ *routing-action rr* = *act* ∧
(∀ *ra* ∈ *set rtbl*. *routing-rule-sort-key ra* < *routing-rule-sort-key rr* ⟶ ¬*prefix-match-semantics* (*routing-match ra*) *addr*))
proof(*induction rtbl*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons rr rtbl*)
show *?case* **proof**(*cases prefix-match-semantics* (*routing-match rr*) *addr*)
case *False*
hence [*simp*]: *routing-table-semantics* (*rr* # *rtbl*) *addr* = *routing-table-semantics* (*rr* # *rtbl*) *addr* **by** *simp*
show *?thesis* **proof**(*cases routing-prefix rr* = *0*)
case *True*

Need special treatment, rtbl won't have a default route, so the IH is not usable.

have *valid-prefix* (*routing-match rr*) **using** *Cons.prems valid-prefixes-split* **by** *blast*
with *True False* **have** *False* **using** *zero-prefix-match-all* **by** *blast*
thus *?thesis* **..**
next
case *False*
with *Cons.prems* **have** *mprems*: *valid-prefixes rtbl is-longest-prefix-routing rtbl has-default-route rtbl unambiguous-routing rtbl*
by(*simp-all add*: *valid-prefixes-split unambiguous-routing-Cons is-longest-prefix-routing-def*)
show *?thesis* **using** *Cons.IH*[*OF mprems*] *False* ‹¬ *prefix-match-semantics* (*routing-match rr*) *addr*› **by** *simp*
qed
next
case *True*
from *True* **have** [*simp*]: *routing-table-semantics* (*rr* # *rtbl*) *addr* = *routing-action rr* **by** *simp*
show *?thesis* (**is** *?l* ⟷ *?r*) **proof**
assume *?l*
hence [*simp*]: *act* = *routing-action rr* **by**(*simp add*: *True*)
have ∗: (∀ *ra*∈*set* (*rr* # *rtbl*). *routing-rule-sort-key rr* ≤ *routing-rule-sort-key ra*)
using ‹*is-longest-prefix-routing* (*rr* # *rtbl*)› **by**(*clarsimp simp*: *is-longest-prefix-routing-def*)

thus *?r* **by**(*fastforce simp add*: *True*)
next
assume *?r*
then guess *rr′* **.. note** *rr′* = *this*
have *rr′* = *rr* **proof**(*rule ccontr*)
assume *C*: *rr′* ≠ *rr*
from *C* **have** *e*: *rr′* ∈ *set rtbl* **using** *rr′* **by** *simp*
show *False* **proof** *cases*
assume *eq*: *routing-match rr′* = *routing-match rr*
with *e* **have** *routing-rule-sort-key rr* < *routing-rule-sort-key rr′* **using** *unambigous-prefix-routing-strong-mono*[*OF Cons.prems*(*2*,*4*) - *eq*] **by** *simp*
with *True rr′* **show** *False* **by** *simp*
next
assume *ne*: *routing-match rr′* ≠ *routing-match rr*
from *rr′ Cons.prems* **have** *valid-prefix* (*routing-match rr*) *valid-prefix* (*routing-match rr′*) *prefix-match-semantics* (*routing-match rr′*) *addr* **by**(*auto simp add*: *valid-prefixes-alt-def*)
note *same-length-prefixes-distinct*[*OF this*(*1*,*2*) *ne*[*symmetric*] - *True this*(*3*)]
moreover have *routing-prefix rr* = *routing-prefix rr′* (**is** *?pe*) **proof** −
have *routing-rule-sort-key rr* < *routing-rule-sort-key rr′* ⟶ ¬ *prefix-match-semantics* (*routing-match rr*) *addr* **using** *rr′* **by** *simp*
with *unambigous-prefix-routing-weak-mono*[*OF Cons.prems*(*2*) *e*] *True* **have** *routing-rule-sort-key rr* = *routing-rule-sort-key rr′* **by** *simp*
thus *?pe* **by**(*simp add*: *routing-rule-sort-key-def int-of-nat-def*)
qed
ultimately show *False* **.**
qed
qed
thus *?l* **using** *rr′* **by** *simp*
qed
qed
qed

1.4 Printing

definition *routing-rule-32-toString* (*rr*::*32 routing-rule*) ≡
prefix-match-32-toString (*routing-match rr*)
@ (*case next-hop* (*routing-action rr*) *of Some nh* ⇒ ″ *via* ″ @ *ipv4addr-toString nh* | - ⇒ [])
@ ″ *dev* ″ @ *routing-oiface rr*
@ ″ *metric* ″ @ *string-of-nat* (*metric rr*)

definition *routing-rule-128-toString* (*rr*::*128 routing-rule*) ≡
prefix-match-128-toString (*routing-match rr*)
@ (*case next-hop* (*routing-action rr*) *of Some nh* ⇒ ″ *via* ″ @ *ipv6addr-toString nh* | - ⇒ [])
@ ″ *dev* ″ @ *routing-oiface rr*
@ ″ *metric* ″ @ *string-of-nat* (*metric rr*)

lemma *map routing-rule-32-toString*
[rr-ctor (42,0,0,0) 7 ''eth0'' None 808,
rr-ctor (0,0,0,0) 0 ''eth1'' (Some (222,173,190,239)) 707] =
[''42.0.0.0/7 dev eth0 metric 808'',
''0.0.0.0/0 via 222.173.190.239 dev eth1 metric 707''] **by** *eval*

2 Routing table to Relation

Walking through a routing table splits the (remaining) IP space when traversing a routing table into a pair of sets: the pair contains the IPs concerned by the current rule and those left alone.

private definition *ipset-prefix-match* **where**
ipset-prefix-match pfx rg = (let pfxrg = prefix-to-wordset pfx in (rg ∩ pfxrg, rg − pfxrg))
private lemma *ipset-prefix-match-m[simp]: fst (ipset-prefix-match pfx rg) = rg ∩ (prefix-to-wordset pfx)* **by**(*simp only: Let-def ipset-prefix-match-def, simp*)
private lemma *ipset-prefix-match-nm[simp]: snd (ipset-prefix-match pfx rg) = rg − (prefix-to-wordset pfx)* **by**(*simp only: Let-def ipset-prefix-match-def, simp*)
private lemma *ipset-prefix-match-distinct: rpm = ipset-prefix-match pfx rg ⟹ (fst rpm) ∩ (snd rpm) = {}* **by** *force*
private lemma *ipset-prefix-match-complete: rpm = ipset-prefix-match pfx rg ⟹ (fst rpm) ∪ (snd rpm) = rg* **by** *force*
private lemma *rpm-m-dup-simp: rg ∩ fst (ipset-prefix-match (routing-match r) rg) = fst (ipset-prefix-match (routing-match r) rg)*
by *simp*
private definition *range-prefix-match :: 'i::len prefix-match ⇒ 'i wordinterval ⇒ 'i wordinterval × 'i wordinterval* **where**
range-prefix-match pfx rg ≡ (let pfxrg = prefix-to-wordinterval pfx in
(wordinterval-intersection rg pfxrg, wordinterval-setminus rg pfxrg))
private lemma *range-prefix-match-set-eq:*
(λ(r1,r2). (wordinterval-to-set r1, wordinterval-to-set r2)) (range-prefix-match pfx rg) =
ipset-prefix-match pfx (wordinterval-to-set rg)
unfolding *range-prefix-match-def ipset-prefix-match-def Let-def*
using *wordinterval-intersection-set-eq wordinterval-setminus-set-eq prefix-to-wordinterval-set-eq*
by *auto*
private lemma *range-prefix-match-sm[simp]: wordinterval-to-set (fst (range-prefix-match pfx rg)) =*
fst (ipset-prefix-match pfx (wordinterval-to-set rg))
by (*metis fst-conv ipset-prefix-match-m wordinterval-intersection-set-eq prefix-to-wordinterval-set-eq range-prefix-match-def*)
private lemma *range-prefix-match-snm[simp]: wordinterval-to-set (snd (range-prefix-match pfx rg)) =*
snd (ipset-prefix-match pfx (wordinterval-to-set rg))
by (*metis snd-conv ipset-prefix-match-nm wordinterval-setminus-set-eq prefix-to-wordinterval-set-eq range-prefix-match-def*)

2.1 Wordintervals for Ports by Routing

This split, although rather trivial, can be used to construct the sets (or rather: the intervals) of IPs that are actually matched by an entry in a routing table.

private fun *routing-port-ranges* :: *'i prefix-routing* ⇒ *'i wordinterval* ⇒ (*string* × (*'i*::*len*) *wordinterval*) *list* **where**
routing-port-ranges [] *lo* = (*if wordinterval-empty lo then* [] *else* [(*routing-oiface* (*undefined*::*'i routing-rule*),*lo*)]) |
routing-port-ranges (*a#as*) *lo* = (
 let rpm = *range-prefix-match* (*routing-match a*) *lo*; *m* = *fst rpm*; *nm* = *snd rpm in* (
 (*routing-oiface a*,*m*) # *routing-port-ranges as nm*))

private lemma *routing-port-ranges-subsets*:
(*a1*, *b1*) ∈ *set* (*routing-port-ranges tbl s*) ⟹ *wordinterval-to-set b1* ⊆ *wordinterval-to-set s*
 by(*induction tbl arbitrary*: *s*; *fastforce simp add*: *Let-def split*: *if-splits*)

private lemma *routing-port-ranges-sound*: *e* ∈ *set* (*routing-port-ranges tbl s*) ⟹
k ∈ *wordinterval-to-set* (*snd e*) ⟹ *valid-prefixes tbl* ⟹
 fst e = *output-iface* (*routing-table-semantics tbl k*)
proof(*induction tbl arbitrary*: *s*)
 case (*Cons a as*)
 note *s* = *Cons.prems*(*1*)[*unfolded routing-port-ranges.simps Let-def list.set*]
 note *vpfx* = *valid-prefixes-split*[*OF Cons.prems*(*3*)]
 show *?case* (**is** *?kees*) **proof**(*cases e* = (*routing-oiface a*, *fst* (*range-prefix-match* (*routing-match a*) *s*)))
  case *False*
  hence *es*: *e* ∈ *set* (*routing-port-ranges as* (*snd* (*range-prefix-match* (*routing-match a*) *s*))) **using** *s* **by** *blast*
  note *eq* = *Cons.IH*[*OF this Cons.prems*(*2*) *conjunct2*[*OF vpfx*]]
  have ¬*prefix-match-semantics* (*routing-match a*) *k* (**is** *?nom*)
  proof −
   from *routing-port-ranges-subsets*[*of fst e snd e*, *unfolded prod.collapse*, *OF es*]
   have ∗: *wordinterval-to-set* (*snd e*) ⊆ *wordinterval-to-set* (*snd* (*range-prefix-match* (*routing-match a*) *s*)) **.**
   show *?nom* **unfolding** *prefix-match-semantics-wordset*[*OF conjunct1*[*OF vpfx*]]
    using ∗ *Cons.prems*(*2*) **unfolding** *wordinterval-subset-set-eq*
    by(*auto simp add*: *range-prefix-match-def Let-def*)
  qed
  thus *?kees* **using** *eq* **by** *simp*
 next
  case *True*
  hence *fe*: *fst e* = *routing-oiface a* **by** *simp*
  from *True* **have** *k* ∈ *wordinterval-to-set* (*fst* (*range-prefix-match* (*routing-match a*) *s*))
   using *Cons.prems*(*2*) **by**(*simp*)
  hence *prefix-match-semantics* (*routing-match a*) *k*

unfolding *prefix-match-semantics-wordset*[*OF conjunct1*, *OF vpfx*]
unfolding *range-prefix-match-def Let-def*
by *simp*
thus *?kees* **by**(*simp add*: *fe*)
qed
qed (*simp split*: *if-splits*)

private lemma *routing-port-ranges-disjoined*:
assumes *vpfx*: *valid-prefixes tbl*
and *ins*: $(a1, b1) \in set$ (*routing-port-ranges tbl s*) $(a2, b2) \in set$ (*routing-port-ranges tbl s*)
and *nemp*: *wordinterval-to-set b1* $\neq \{\}$
shows $b1 \neq b2 \longleftrightarrow$ *wordinterval-to-set b1* $\cap$ *wordinterval-to-set b2* $= \{\}$
using *assms*
proof(*induction tbl arbitrary*: *s*)
case (*Cons r rs*)
have *vpfx*: *valid-prefix* (*routing-match r*) *valid-prefixes rs* **using** *Cons.prems*(*1*) **using** *valid-prefixes-split* **by** *blast+*
{
fix *a1 b1 a2 b2*
assume *one*: *b1* = *fst* (*range-prefix-match* (*routing-match r*) *s*)
assume *two*: $(a2, b2) \in$ *set* (*routing-port-ranges rs* (*snd* (*range-prefix-match* (*routing-match r*) *s*)))
have *dc*: *wordinterval-to-set* (*snd* (*range-prefix-match* (*routing-match r*) *s*)) $\cap$
wordinterval-to-set (*fst* (*range-prefix-match* (*routing-match r*) *s*)) $= \{\}$ **by** *force*
hence *wordinterval-to-set b1* $\cap$ *wordinterval-to-set b2* $= \{\}$
unfolding *one* **using** *two*[*THEN routing-port-ranges-subsets*] **by** *fast*
} **note** $*$ = *this*
show *?case*
using ‹$(a1, b1) \in set$ (*routing-port-ranges* (*r # rs*) *s*)› ‹$(a2, b2) \in set$ (*routing-port-ranges* (*r # rs*) *s*)› *nemp*
Cons.IH[*OF vpfx*(*2*)] $*$
by(*fastforce simp add*: *Let-def*)
qed (*simp split*: *if-splits*)

private lemma *routing-port-rangesI*:
valid-prefixes tbl $\Longrightarrow$
output-iface (*routing-table-semantics tbl k*) = *output-port* $\Longrightarrow$
$k \in$ *wordinterval-to-set wi* $\Longrightarrow$
($\exists$ *ip-range*. (*output-port*, *ip-range*) $\in$ *set* (*routing-port-ranges tbl wi*) $\wedge$ $k \in$ *wordinterval-to-set ip-range*)
proof(*induction tbl arbitrary*: *wi*)
case *Nil* **thus** *?case* **by** *simp blast*
next
case (*Cons r rs*)
from *Cons.prems*(*1*) **have** *vpfx*: *valid-prefix* (*routing-match r*) **and** *vpfxs*: *valid-prefixes rs*
by(*simp-all add*: *valid-prefixes-split*)

show *?case*
proof(*cases prefix-match-semantics* (*routing-match r*) *k*)
case *True*
thus *?thesis*
using *Cons.prems*(*2*) **using** *vpfx* ‹*k* ∈ *wordinterval-to-set wi*›
by (*intro exI*[**where** *x* = *fst* (*range-prefix-match* (*routing-match r*) *wi*)])
(*simp add*: *prefix-match-semantics-wordset Let-def*)
next
case *False*
with ‹*k* ∈ *wordinterval-to-set wi*› **have** *ksnd*: *k* ∈ *wordinterval-to-set* (*snd* (*range-prefix-match* (*routing-match r*) *wi*))
by (*simp add*: *prefix-match-semantics-wordset vpfx*)
have *mpr*: *output-iface* (*routing-table-semantics rs k*) = *output-port* **using** *Cons.prems False* **by** *simp*
note *Cons.IH*[*OF vpfxs mpr ksnd*]
thus *?thesis* **by**(*fastforce simp*: *Let-def*)
qed
qed

2.2 Reduction

So far, one entry in the list would be generated for each routing table entry. This next step reduces it to one for each port. The resulting list will represent a function from port to IP wordinterval. (It can also be understood as a function from IP (interval) to port (where the intervals don't overlap).

definition *reduce-range-destination l* ≡
let ps = *remdups* (*map fst l*) *in*
let c = *λs.* (*wordinterval-Union* ∘ *map snd* ∘ *filter* (((=) *s*) ∘ *fst*)) *l in*
[(*p*, *c p*). *p* ← *ps*]

definition *routing-ipassmt-wi tbl* ≡ *reduce-range-destination* (*routing-port-ranges tbl wordinterval-UNIV*)

lemma *routing-ipassmt-wi-distinct*: *distinct* (*map fst* (*routing-ipassmt-wi tbl*))
unfolding *routing-ipassmt-wi-def reduce-range-destination-def*
by(*simp add*: *comp-def*)

private lemma *routing-port-ranges-superseted*:
(*a1*,*b1*) ∈ *set* (*routing-port-ranges tbl wordinterval-UNIV*) ⟹
∃ *b2*. (*a1*,*b2*) ∈ *set* (*routing-ipassmt-wi tbl*) ∧ *wordinterval-to-set b1* ⊆ *wordinterval-to-set b2*
unfolding *routing-ipassmt-wi-def reduce-range-destination-def*
by(*force simp add*: *Set.image-iff wordinterval-Union*)

private lemma *routing-ipassmt-wi-subsetted*:
(*a1*,*b1*) ∈ *set* (*routing-ipassmt-wi tbl*) ⟹
(*a1*,*b2*) ∈ *set* (*routing-port-ranges tbl wordinterval-UNIV*) ⟹ *wordinterval-to-set*

b2 ⊆ *wordinterval-to-set b1*
unfolding *routing-ipassmt-wi-def reduce-range-destination-def*
by(*fastforce simp add*: *Set.image-iff wordinterval-Union comp-def*)

This lemma should hold without the *valid-prefixes* assumption, but that would break the semantic argument and make the proof a lot harder.

lemma *routing-ipassmt-wi-disjoint*:
assumes *vpfx*: *valid-prefixes* (*tbl*::('*i*::*len*) *prefix-routing*)
and *dif*: *a1* ≠ *a2*
and *ins*: (*a1*, *b1*) ∈ *set* (*routing-ipassmt-wi tbl*) (*a2*, *b2*) ∈ *set* (*routing-ipassmt-wi tbl*)
shows *wordinterval-to-set b1* ∩ *wordinterval-to-set b2* = {}
proof(*rule ccontr*)
note *iuf* = *ins*[*unfolded routing-ipassmt-wi-def reduce-range-destination-def Let-def*, *simplified*, *unfolded Set.image-iff comp-def*, *simplified*]
assume (*wordinterval-to-set b1* ∩ *wordinterval-to-set b2* ≠ {})
hence *wordinterval-to-set b1* ∩ *wordinterval-to-set b2* ≠ {} **by** *simp*

If the intervals are not disjoint, there exists a witness of that.

then obtain *x* **where** *x*[*simp*]: *x* ∈ *wordinterval-to-set b1 x* ∈ *wordinterval-to-set b2* **by** *blast*

This witness has to have come from some entry in the result of *routing-port-ranges*, for both of *b1* and *b2*.

hence ∃ *b1g*. *x* ∈ *wordinterval-to-set b1g* ∧ *wordinterval-to-set b1g* ⊆ *wordinterval-to-set b1* ∧ (*a1*, *b1g*) ∈ *set* (*routing-port-ranges tbl wordinterval-UNIV*)
using *iuf*(*1*) **by**(*fastforce simp add*: *wordinterval-Union*)
then obtain *b1g* **where** *b1g*: *x* ∈ *wordinterval-to-set b1g wordinterval-to-set b1g* ⊆ *wordinterval-to-set b1* (*a1*, *b1g*) ∈ *set* (*routing-port-ranges tbl wordinterval-UNIV*) **by** *clarsimp*
from *x* **have** ∃ *b2g*. *x* ∈ *wordinterval-to-set b2g* ∧ *wordinterval-to-set b2g* ⊆ *wordinterval-to-set b2* ∧ (*a2*, *b2g*) ∈ *set* (*routing-port-ranges tbl wordinterval-UNIV*)
using *iuf*(*2*) **by**(*fastforce simp add*: *wordinterval-Union*)
then obtain *b2g* **where** *b2g*: *x* ∈ *wordinterval-to-set b2g wordinterval-to-set b2g* ⊆ *wordinterval-to-set b2* (*a2*, *b2g*) ∈ *set* (*routing-port-ranges tbl wordinterval-UNIV*) **by** *clarsimp*

Soudness tells us that the both *a1* and *a2* have to be the result of routing *x*.

note *routing-port-ranges-sound*[*OF b1g*(*3*), *unfolded fst-conv snd-conv*, *OF b1g*(*1*) *vpfx*] *routing-port-ranges-sound*[*OF b2g*(*3*), *unfolded fst-conv snd-conv*, *OF b2g*(*1*) *vpfx*]

A contradiction follows from *a1* ≠ *a2*.

with *dif* **show** *False* **by** *simp*
qed

lemma *routing-ipassmt-wi-sound*:

assumes *vpfx*: *valid-prefixes tbl*
and *ins*: (*ea*,*eb*) ∈ *set* (*routing-ipassmt-wi tbl*)
and *x*: *k* ∈ *wordinterval-to-set eb*
shows *ea* = *output-iface* (*routing-table-semantics tbl k*)
proof −
note *iuf* = *ins*[*unfolded routing-ipassmt-wi-def reduce-range-destination-def Let-def*, *simplified*, *unfolded Set.image-iff comp-def*, *simplified*]
from *x* **have** ∃ *b1g*. *k* ∈ *wordinterval-to-set b1g* ∧ *wordinterval-to-set b1g* ⊆ *wordinterval-to-set eb* ∧ (*ea*, *b1g*) ∈ *set* (*routing-port-ranges tbl wordinterval-UNIV*)
using *iuf*(*1*) **by**(*fastforce simp add*: *wordinterval-Union*)
then obtain *b1g* **where** *b1g*: *k* ∈ *wordinterval-to-set b1g wordinterval-to-set b1g* ⊆ *wordinterval-to-set eb* (*ea*, *b1g*) ∈ *set* (*routing-port-ranges tbl wordinterval-UNIV*) **by** *clarsimp*
note *routing-port-ranges-sound*[*OF b1g*(*3*), *unfolded fst-conv snd-conv*, *OF b1g*(*1*) *vpfx*]
thus *?thesis* **.**
qed

theorem *routing-ipassmt-wi*:
assumes *vpfx*: *valid-prefixes tbl*
shows
output-iface (*routing-table-semantics tbl k*) = *output-port* ⟷
(∃ *ip-range*. *k* ∈ *wordinterval-to-set ip-range* ∧ (*output-port*, *ip-range*) ∈ *set* (*routing-ipassmt-wi tbl*))
proof (*intro iffI*, *goal-cases*)
case *2* **with** *vpfx routing-ipassmt-wi-sound* **show** *?case* **by** *blast*
next
case *1*
then obtain *ip-range* **where** (*output-port*, *ip-range*) ∈ *set* (*routing-port-ranges tbl wordinterval-UNIV*) ∧ *k* ∈ *wordinterval-to-set ip-range*
using *routing-port-rangesI*[**where** *wi* = *wordinterval-UNIV*, *OF vpfx*] **by** *auto*
thus *?case*
unfolding *routing-ipassmt-wi-def reduce-range-destination-def*
unfolding *Let-def comp-def*
by(*force simp add*: *Set.image-iff wordinterval-Union*)
qed

lemma *routing-ipassmt-wi-has-all-interfaces*:
assumes *in-tbl*: *r* ∈ *set tbl*
shows ∃ *s*. (*routing-oiface r*,*s*) ∈ *set* (*routing-ipassmt-wi tbl*)
proof −
from *in-tbl* **have** ∃ *s*. (*routing-oiface r*,*s*) ∈ *set* (*routing-port-ranges tbl S*) **for** *S*
proof(*induction tbl arbitrary*: *S*)
case (*Cons l ls*)
show *?case*
proof(*cases r* = *l*)
case *True* **thus** *?thesis* **using** *Cons.prems* **by**(*auto simp*: *Let-def*)
next

case *False* **with** *Cons.prems* **have** $r \in set\ ls$ **by** *simp*
from *Cons.IH*[*OF this*] **show** *?thesis* **by**(*simp add*: *Let-def*) *blast*
qed
qed *simp*
thus *?thesis*
unfolding *routing-ipassmt-wi-def reduce-range-destination-def*
by(*force simp add*: *Set.image-iff*)
qed

end

end

3 Linux Router

theory *Linux-Router*
imports
Routing-Table
Simple-Firewall.SimpleFw-Semantics
Simple-Firewall.Simple-Packet
HOL−Library.Monad-Syntax
begin

definition *fromMaybe a m* = (*case m of Some a* ⇒ *a* | *None* ⇒ *a*)

Here, we present a heavily simplified model of a linux router. (i.e., a linux-based device with `net.ipv4.ip_forward`) It covers the following steps in packet processing:

- Packet arrives (destination port is empty, destination mac address is own address).
- Destination address is extracted and used for a routing table lookup.
- Packet is updated with output interface of routing decision.
- The FORWARD chain of iptables is considered.
- Next hop is extracted from the routing decision, fallback to destination address if directly attached.
- MAC address of next hop is looked up (using the mac lookup function mlf)
- L2 destination address of packet is updated.

This is stripped down to model only the most important and widely used aspects of packet processing. Here are a few examples of what was abstracted away:

- No local traffic.
- Only the `filter` table of iptables is considered, `raw` and `nat` are not.
- Only one routing table is considered. (Linux can have other tables than the `default` one.)
- No source MAC modification.
- …

record *interface* =
iface-name :: *string*
iface-mac :: *48 word*

definition *iface-packet-check* :: *interface list* ⇒('*i*::*len*,'*b*) *simple-packet-ext-scheme* ⇒ *interface option*
where *iface-packet-check ifs p* ≡ *find* (λ*i*. *iface-name i* = *p-iiface p* ∧ *iface-mac i* = *p-l2dst p*) *ifs*
term *simple-fw*
definition *simple-linux-router* ::
'*i routing-rule list* ⇒ '*i simple-rule list* ⇒ (('*i*::*len*) *word* ⇒ *48 word option*) ⇒ *interface list* ⇒ '*i simple-packet-ext* ⇒ '*i simple-packet-ext option* **where**
simple-linux-router rt fw mlf ifl p ≡ *do* {
\- ← *iface-packet-check ifl p*;
let rd — (routing decision) = *routing-table-semantics rt* (*p-dst p*);
let p = *p*(|*p-oiface* := *output-iface rd*|);
let fd — (firewall decision) = *simple-fw fw p*;
\- ← (*case fd of Decision FinalAllow* ⇒ *Some* () | *Decision FinalDeny* ⇒ *None*);
let nh = *fromMaybe* (*p-dst p*) (*next-hop rd*);
ma ← *mlf nh*;
Some (*p*(|*p-l2dst* := *ma*|))
}

However, the above model is still too powerful for some use-cases. Especially, the next hop look-up cannot be done without either a pre-distributed table of all MAC addresses, or the usual mechanic of sending out an ARP request and caching the answer. Doing ARP requests in the restricted environment of, e.g., an OpenFlow ruleset seems impossible. Therefore, we present this model:

definition *simple-linux-router-nol12* ::
'*i routing-rule list* ⇒ '*i simple-rule list* ⇒ ('*i*,'*a*) *simple-packet-scheme* ⇒ ('*i*::*len*,'*a*) *simple-packet-scheme option* **where**
simple-linux-router-nol12 rt fw p ≡ *do* {
let rd = *routing-table-semantics rt* (*p-dst p*);
let p = *p*(|*p-oiface* := *output-iface rd*|);
let fd = *simple-fw fw p*;
\- ← (*case fd of Decision FinalAllow* ⇒ *Some* () | *Decision FinalDeny* ⇒ *None*);

Some p
}

The differences to *simple-linux-router* are illustrated by the lemmata below.

lemma *rtr-nomac-e1*:
fixes *pi*
assumes *simple-linux-router rt fw mlf ifl pi = Some po*
assumes *simple-linux-router-nol12 rt fw pi = Some po'*
shows $\exists$ *x. po = po'*⦇*p-l2dst := x*⦈
using *assms*
unfolding *simple-linux-router-nol12-def simple-linux-router-def*
by(*simp add*: *Let-def split*: *option.splits state.splits final-decision.splits Option.bind-splits if-splits*) *blast+*

lemma *rtr-nomac-e2*:
fixes *pi*
assumes *simple-linux-router rt fw mlf ifl pi = Some po*
shows $\exists$ *po'. simple-linux-router-nol12 rt fw pi = Some po'*
using *assms*
unfolding *simple-linux-router-nol12-def simple-linux-router-def*
by(*clarsimp simp add*: *Let-def split*: *option.splits state.splits final-decision.splits Option.bind-splits if-splits*)

lemma *rtr-nomac-e3*:
fixes *pi*
assumes *simple-linux-router-nol12 rt fw pi = Some po*
assumes *iface-packet-check ifl pi = Some i* — don't care
assumes *mlf* (*fromMaybe* (*p-dst pi*) (*next-hop* (*routing-table-semantics rt* (*p-dst pi*)))) = *Some i2*
shows $\exists$ *po'. simple-linux-router rt fw mlf ifl pi = Some po'*
using *assms*
unfolding *simple-linux-router-nol12-def simple-linux-router-def*
by(*clarsimp simp add*: *Let-def split*: *option.splits state.splits final-decision.splits Option.bind-splits if-splits*)

lemma *rtr-nomac-eq*:
fixes *pi*
assumes *iface-packet-check ifl pi* $\neq$ *None*
assumes *mlf* (*fromMaybe* (*p-dst pi*) (*next-hop* (*routing-table-semantics rt* (*p-dst pi*)))) $\neq$ *None*
shows $\exists$ *x. map-option* (λ*p. p*⦇*p-l2dst := x*⦈) (*simple-linux-router-nol12 rt fw pi*) = *simple-linux-router rt fw mlf ifl pi*
proof(*cases simple-linux-router-nol12 rt fw pi*; *cases simple-linux-router rt fw mlf ifl pi*)
fix *a b*
assume *as*: *simple-linux-router rt fw mlf ifl pi = Some b simple-linux-router-nol12 rt fw pi = Some a*
note *rtr-nomac-e1*[*OF this*]
with *as* **show** *?thesis* **by** *auto*

next
fix *a* **assume** *as*: *simple-linux-router-nol12 rt fw pi* = *None simple-linux-router rt fw mlf ifl pi* = *Some a*
note *rtr-nomac-e2*[*OF as*(*2*)]
with *as*(*1*) **have** *False* **by** *simp*
thus *?thesis* **..**
next
fix *a* **assume** *as*: *simple-linux-router-nol12 rt fw pi* = *Some a simple-linux-router rt fw mlf ifl pi* = *None*
from ‹*iface-packet-check ifl pi* ≠ *None*› **obtain** *i3* **where** *iface-packet-check ifl pi* = *Some i3* **by** *blast*
note *rtr-nomac-e3*[*OF as*(*1*) *this*] *assms*(*2*)
with *as*(*2*) **have** *False* **by** *force*
thus *?thesis* **..**
qed *simp*

end

4 Parser

theory *IpRoute-Parser*
imports *Routing-Table*
IP-Addresses.IP-Address-Parser
keywords *parse-ip-route parse-ip-6-route* :: *thy-decl*
begin

This helps to read the output of the `ip route` command into a *32 routing-rule list*.

definition *empty-rr-hlp* :: (*'a*::*len*) *prefix-match* ⇒ *'a routing-rule* **where**
empty-rr-hlp pm = *routing-rule.make pm default-metric* (*routing-action.make* '''' *None*)

lemma *empty-rr-hlp-alt*:
empty-rr-hlp pm = (| *routing-match* = *pm*, *metric* = *0*, *routing-action* = (|*output-iface* = [], *next-hop* = *None*|)|)
unfolding *empty-rr-hlp-def routing-rule.defs default-metric-def routing-action.defs* **..**

definition *routing-action-next-hop-update* :: *'a word* ⇒ *'a routing-rule* ⇒ (*'a*::*len*) *routing-rule*
where
routing-action-next-hop-update h pk = *pk*(| *routing-action* := (*routing-action pk*)(| *next-hop* := *Some h*|) |)
lemma *routing-action-next-hop-update h pk* = *routing-action-update* (*next-hop-update* (λ-. (*Some h*))) (*pk*::*32 routing-rule*)
by(*simp add*: *routing-action-next-hop-update-def*)

definition *routing-action-oiface-update* :: *string* ⇒ *'a routing-rule* ⇒ (*'a*::*len*) *routing-rule*

where
routing-action-oiface-update h pk = *routing-action-update* (*output-iface-update* (λ-. *h*)) (*pk*::′*a routing-rule*)
lemma *routing-action-oiface-update h pk* = *pk*⦇ *routing-action* := (*routing-action pk*)⦇ *output-iface* := *h*⦈ ⦈
by(*simp add*: *routing-action-oiface-update-def*)

definition *default-prefix* = *PrefixMatch 0 0*
lemma *default-prefix-matchall*: *prefix-match-semantics default-prefix ip*
unfolding *default-prefix-def* **by** (*simp add*: *valid-prefix-00 zero-prefix-match-all*)

definition *sanity-ip-route* (*r*::(′*a*::*len*) *prefix-routing*) ≡ *correct-routing r* ∧ *unambiguous-routing r* ∧ *list-all* ((≠) ′′′′ ∘ *routing-oiface*) *r*

The parser ensures that *sanity-ip-route* holds for any ruleset that is imported.

ML-file ‹*IpRoute-Parser.ML*›

ML‹
Outer-Syntax.local-theory @{*command-keyword parse-ip-route*}
Load a file generated by ip route and make the routing table definition available as isabelle term
(*Parse.binding* −−| @{*keyword* =} −− *Parse.string* >> *register-ip-route 32*)
›

ML‹
Outer-Syntax.local-theory @{*command-keyword parse-ip-6-route*}
Load a file generated by ip −*6 route and make the routing table definition available as isabelle term*
(*Parse.binding* −−| @{*keyword* =} −− *Parse.string* >> *register-ip-route 128*)
›

parse-ip-route *rtbl-parser-test1* = *ip*−*route*−*ex*
lemma *sanity-ip-route rtbl-parser-test1* **by** *eval*

lemma *rtbl-parser-test1* =
[⦇*routing-match* = *PrefixMatch 0xFFFFFF00 32*, *metric* = *0*, *routing-action* = ⦇*output-iface* = ′′*tun0*′′, *next-hop* = *None*⦈⦈,
⦇*routing-match* = *PrefixMatch 0xA0D2AA0 28*, *metric* = *303*, *routing-action* = ⦇*output-iface* = ′′*ewlan*′′, *next-hop* = *None*⦈⦈,
⦇*routing-match* = *PrefixMatch 0xA0D2500 24*, *metric* = *0*, *routing-action* = ⦇*output-iface* = ′′*tun0*′′, *next-hop* = *Some 0xFFFFFF00*⦈⦈,
⦇*routing-match* = *PrefixMatch 0xA0D2C00 24*, *metric* = *0*, *routing-action* = ⦇*output-iface* = ′′*tun0*′′, *next-hop* = *Some 0xFFFFFF00*⦈⦈,
⦇*routing-match* = *PrefixMatch 0 0*, *metric* = *303*, *routing-action* = ⦇*output-iface* = ′′*ewlan*′′, *next-hop* = *Some 0xA0D2AA1*⦈⦈]
by *eval*

parse-ip-6-route *rtbl-parser-test2* = *ip*−*6*−*route*−*ex*
value[*code*] *rtbl-parser-test2*
lemma *sanity-ip-route rtbl-parser-test2* **by** *eval*

end