Routing

Julius Michaelis, Cornelius Diekmann

June 16, 2019

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* ≡ (*case a of LinordHelper a1 a2* ⇒ *case b of LinordHelper b1 b2* ⇒ *a1* < *b1* ∨ *a1* = *b1* ∧ *a2* ≤ *b2*)
definition *a* ≤ *b* ⟷ *linord-helper-less-eq1 a b*
definition *a* < *b* ⟷ (*a* ≠ *b* ∧ *linord-helper-less-eq1 a b*)
instance
⟨*proof*⟩
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
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 (λrr. valid-prefix (routing-match rr)) r) True
lemma *valid-prefixes-split*: *valid-prefixes (r#rs) ⟹ valid-prefix (routing-match r) ∧ valid-prefixes rs*
⟨*proof*⟩

lemma *foldr-True-set*: *foldr (λx. (∧) (f x)) l True = (∀ x ∈ set l. f x)*
⟨*proof*⟩
lemma *valid-prefixes-alt-def*: *valid-prefixes r = (∀ e ∈ set r. valid-prefix (routing-match e))*
⟨*proof*⟩

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)* ⟨*proof*⟩

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*⟩

1.3 Longest Prefix Match

We can abuse *LinordHelper* to sort.

definition *routing-rule-sort-key* ≡ λ*r*. *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*)
⟨*proof*⟩

lemma *int-of-nat-less*: *int-of-nat a* < *int-of-nat b* ⟹ *a* < *b* ⟨*proof*⟩

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*))
⟨*proof*⟩

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*) ⟨*proof*⟩

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*
⟨*proof*⟩
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*
⟨*proof*⟩
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*⟩

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*
⟨*proof*⟩
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*⟩

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*) ⟨*proof*⟩

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*⟩

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''] ⟨*proof*⟩

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*) ⟨*proof*⟩ **lemma** *ipset-prefix-match-nm*[*simp*]: *snd* (*ipset-prefix-match pfx rg*) = *rg* − (*prefix-to-wordset pfx*) ⟨*proof*⟩ **lemma** *ipset-prefix-match-distinct*: *rpm* = *ipset-prefix-match pfx rg* ⟹
(*fst rpm*) ∩ (*snd rpm*) = {} ⟨*proof*⟩ **lemma** *ipset-prefix-match-complete*: *rpm* = *ipset-prefix-match pfx rg* ⟹
(*fst rpm*) ∪ (*snd rpm*) = *rg* ⟨*proof*⟩ **lemma** *rpm-m-dup-simp*: *rg* ∩ *fst* (*ipset-prefix-match* (*routing-match r*) *rg*) = *fst* (*ipset-prefix-match* (*routing-match r*) *rg*)
⟨*proof*⟩ **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*)
⟨*proof*⟩ **lemma** *range-prefix-match-sm*[*simp*]: *wordinterval-to-set* (*fst* (*range-prefix-match pfx rg*)) =

fst (*ipset-prefix-match pfx* (*wordinterval-to-set rg*))
⟨*proof*⟩ **lemma** *range-prefix-match-snm*[*simp*]: *wordinterval-to-set* (*snd* (*range-prefix-match pfx rg*)) =
snd (*ipset-prefix-match pfx* (*wordinterval-to-set rg*))
⟨*proof*⟩

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*
⟨*proof*⟩ **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*⟩ **lemma** *routing-port-ranges-disjoined*:
assumes *vpfx*: *valid-prefixes tbl*
and *ins*: (*a1*, *b1*) ∈ *set* (*routing-port-ranges tbl s*) (*a2*, *b2*) ∈ *set* (*routing-port-ranges tbl s*)
and *nemp*: *wordinterval-to-set b1* ≠ {}
shows *b1* ≠ *b2* ⟷ *wordinterval-to-set b1* ∩ *wordinterval-to-set b2* = {}
⟨*proof*⟩ **lemma** *routing-port-rangesI*:
valid-prefixes tbl ⟹
output-iface (*routing-table-semantics tbl k*) = *output-port* ⟹
k ∈ *wordinterval-to-set wi* ⟹
(∃ *ip-range*. (*output-port*, *ip-range*) ∈ *set* (*routing-port-ranges tbl wi*) ∧ *k* ∈ *wordinterval-to-set ip-range*)
⟨*proof*⟩

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))*
⟨*proof*⟩ **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
⟨*proof*⟩ **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
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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|)*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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*⟩

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* $\Rightarrow$ '*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*|)|)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*|) |)
⟨*proof*⟩

definition *default-prefix* = *PrefixMatch 0 0*
lemma *default-prefix-matchall*: *prefix-match-semantics default-prefix ip*
⟨*proof*⟩

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*⟩

parse-ip-route *rtbl-parser-test1* = *ip*−*route*−*ex*
lemma *sanity-ip-route rtbl-parser-test1* ⟨*proof*⟩

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*|)|)]
⟨*proof*⟩

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

lemma *sanity-ip-route rtbl-parser-test2* ⟨*proof*⟩

end