

IP Addresses

Cornelius Diekmann, Julius Michaelis, Lars Hupel

June 14, 2026

Abstract

This entry contains a definition of IP addresses and a library to work with them.

Generic IP addresses are modeled as machine words of arbitrary length. Derived from this generic definition, IPv4 addresses are 32bit machine words, IPv6 addresses are 128bit words. Additionally, IPv4 addresses can be represented in dot-decimal notation and IPv6 addresses in (compressed) colon-separated notation. We support `toString` functions and parsers for both notations. Sets of IP addresses can be represented with a netmask (e.g. `192.168.0.0/255.255.0.0`) or in CIDR notation (e.g. `192.168.0.0/16`). To provide executable code for set operations on IP address ranges, the library includes a datatype to work on arbitrary intervals of machine words.

Contents

1	WordInterval: Executable datatype for Machine Word Sets	1
1.1	Syntax	1
1.2	Semantics	2
1.3	Basic operations	2
1.4	WordInterval and Lists	3
1.5	Optimizing and minimizing <i>'a wordintervals</i>	3
1.6	Further operations	10
2	Definitions inspired by the Haskell World.	18
3	Modelling IP Addresses	19
3.1	Sets of IP Addresses	19
3.2	IP Addresses as WordIntervals	23
3.3	IP Addresses in CIDR Notation	23
3.4	Clever Operations on IP Addresses in CIDR Notation	25
3.5	Code Equations	27

4	IPv4 Addresses	28
4.1	Representing IPv4 Addresses (Syntax)	29
4.2	IP Ranges: Examples	32
5	IPv6 Addresses	34
5.1	Syntax of IPv6 Addresses	35
5.2	Semantics	43
5.3	IPv6 Pretty Printing (converting to compressed format) . . .	50
6	Prefix Match	57
6.1	Address Semantics	59
6.2	Relation between prefix and set	59
6.3	Equivalence Proofs	60
7	CIDR Split Motivation (Example for IPv4)	62
8	CIDR Split	63
8.1	Versions for <i>ipset-from-cidr</i>	70
9	Parsing IP Addresses	71
9.1	IPv4 Parser	71
9.2	IPv6 Parser	72
10	Printing Numbers	74
11	Printing Machine Words	76
12	Printing Lists	79
13	Pretty Printing IP Addresses	80
13.1	Generic Pretty Printer	80
13.2	IPv4 Pretty Printing	80
13.3	IPv6 Pretty Printing	81

```

theory WordInterval
  imports
    Main
    Word-Lib.Word-Lemmas
    Word-Lib.Next-and-Prev
begin

```

1 WordInterval: Executable datatype for Machine Word Sets

Stores ranges of machine words as interval. This has been proven quite efficient for IP Addresses.

1.1 Syntax

```
context
  notes [[typedef-overloaded]]
begin
  datatype ('a::len) wordinterval = WordInterval
    ('a::len) word — start (inclusive)
    ('a::len) word — end (inclusive)
    | RangeUnion 'a wordinterval 'a wordinterval
end
```

1.2 Semantics

```
fun wordinterval-to-set :: 'a::len wordinterval  $\Rightarrow$  ('a::len word) set
where
  wordinterval-to-set (WordInterval start end) =
    {start .. end} |
  wordinterval-to-set (RangeUnion r1 r2) =
    wordinterval-to-set r1  $\cup$  wordinterval-to-set r2
```

1.3 Basic operations

\in

```
fun wordinterval-element :: 'a::len word  $\Rightarrow$  'a::len wordinterval  $\Rightarrow$  bool where
  wordinterval-element el (WordInterval s e)  $\longleftrightarrow$  s  $\leq$  el  $\wedge$  el  $\leq$  e |
  wordinterval-element el (RangeUnion r1 r2)  $\longleftrightarrow$ 
    wordinterval-element el r1  $\vee$  wordinterval-element
```

el r2

```
lemma wordinterval-element-set-eq[simp]:
  wordinterval-element el rg = (el  $\in$  wordinterval-to-set rg)
  by(induction rg rule: wordinterval-element.induct) simp-all
```

definition wordinterval-union

```
:: 'a::len wordinterval  $\Rightarrow$  'a::len wordinterval  $\Rightarrow$  'a::len wordinterval where
  wordinterval-union r1 r2 = RangeUnion r1 r2
```

lemma wordinterval-union-set-eq[simp]:

```
wordinterval-to-set (wordinterval-union r1 r2) = wordinterval-to-set r1  $\cup$  wordinterval-to-set r2
```

unfolding wordinterval-union-def **by** simp

fun wordinterval-empty :: 'a::len wordinterval \Rightarrow bool **where**

```
wordinterval-empty (WordInterval s e)  $\longleftrightarrow$  e < s |
```

$\text{wordinterval-empty } (\text{RangeUnion } r1 \ r2) \longleftrightarrow \text{wordinterval-empty } r1 \wedge \text{wordinterval-empty } r2$

lemma $\text{wordinterval-empty-set-eq}[\text{simp}]$: $\text{wordinterval-empty } r \longleftrightarrow \text{wordinterval-to-set } r = \{\}$

by($\text{induction } r$) **auto**

definition $\text{Empty-WordInterval} :: 'a::\text{len wordinterval}$ **where**

$\text{Empty-WordInterval} \equiv \text{WordInterval } 1 \ 0$

lemma $\text{wordinterval-empty-Empty-WordInterval}$: $\text{wordinterval-empty } \text{Empty-WordInterval}$

by($\text{simp add: Empty-WordInterval-def}$)

lemma $\text{Empty-WordInterval-set-eq}[\text{simp}]$: $\text{wordinterval-to-set } \text{Empty-WordInterval} = \{\}$

by($\text{simp add: Empty-WordInterval-def}$)

1.4 WordInterval and Lists

A list of $(\text{start}, \text{end})$ tuples.

wordinterval to list

fun $\text{wi2l} :: 'a::\text{len wordinterval} \Rightarrow ('a::\text{len word} \times 'a::\text{len word}) \text{ list}$ **where**

$\text{wi2l } (\text{RangeUnion } r1 \ r2) = \text{wi2l } r1 \ @ \ \text{wi2l } r2 \ |$

$\text{wi2l } (\text{WordInterval } s \ e) = (\text{if } e < s \ \text{then } [] \ \text{else } [(s,e)])$

list to wordinterval

fun $\text{l2wi} :: ('a::\text{len word} \times 'a \ \text{word}) \text{ list} \Rightarrow 'a \ \text{wordinterval}$ **where**

$\text{l2wi } [] = \text{Empty-WordInterval} \ |$

$\text{l2wi } [(s,e)] = (\text{WordInterval } s \ e) \ |$

$\text{l2wi } ((s,e)\#rs) = (\text{RangeUnion } (\text{WordInterval } s \ e) \ (\text{l2wi } rs))$

lemma l2wi-append : $\text{wordinterval-to-set } (\text{l2wi } (l1 @ l2)) =$

$\text{wordinterval-to-set } (\text{l2wi } l1) \cup \text{wordinterval-to-set } (\text{l2wi } l2)$

proof($\text{induction } l1$ **arbitrary**: l2 **rule**: l2wi.induct)

case 1 thus $?case$ **by** simp

next

case $(2 \ s \ e \ l2)$ **thus** $?case$ **by** $(\text{cases } l2) \ \text{simp-all}$

next

case 3 thus $?case$ **by** force

qed

lemma $\text{l2wi-wi2l}[\text{simp}]$: $\text{wordinterval-to-set } (\text{l2wi } (\text{wi2l } r)) = \text{wordinterval-to-set } r$

by($\text{induction } r$) $(\text{simp-all add: l2wi-append})$

lemma l2wi : $\text{wordinterval-to-set } (\text{l2wi } l) = (\bigcup (i,j) \in \text{set } l. \{i .. j\})$

by($\text{induction } l$ **rule**: l2wi.induct , simp-all)

lemma wi2l : $(\bigcup (i,j) \in \text{set } (\text{wi2l } r). \{i .. j\}) = \text{wordinterval-to-set } r$

by($\text{induction } r$ **rule**: wi2l.induct , simp-all)

lemma *l2wi-remdups*[simp]: *wordinterval-to-set* (l2wi (remdups ls)) = *wordinterval-to-set* (l2wi ls)
by(simp add: l2wi)

lemma *wi2l-empty*[simp]: *wi2l Empty-WordInterval* = []
unfolding *Empty-WordInterval-def*
by simp

1.5 Optimizing and minimizing 'a wordintervals

Removing empty intervals

context

begin

fun *wordinterval-optimize-empty* :: 'a::len wordinterval \Rightarrow 'a wordinterval **where**
wordinterval-optimize-empty (RangeUnion r1 r2) = (let r1o = *wordinterval-optimize-empty* r1;

r2o = *wordinterval-optimize-empty* r2

in if
wordinterval-empty r1o
then
r2o
else if
wordinterval-empty r2o
then
r1o
else

RangeUnion r1o r2o) |
wordinterval-optimize-empty r = r

lemma *wordinterval-optimize-empty-set-eq*[simp]:
wordinterval-to-set (*wordinterval-optimize-empty* r) = *wordinterval-to-set* r
by(induction r) (simp-all add: Let-def)

lemma *wordinterval-optimize-empty-double*:
wordinterval-optimize-empty (*wordinterval-optimize-empty* r) = *wordinterval-optimize-empty*

r

by(induction r) (simp-all add: Let-def)

private fun *wordinterval-empty-shallow* :: 'a::len wordinterval \Rightarrow bool **where**
wordinterval-empty-shallow (WordInterval s e) \longleftrightarrow e < s |
wordinterval-empty-shallow (RangeUnion - -) \longleftrightarrow False

private lemma *helper-optimize-shallow*:
wordinterval-empty-shallow (*wordinterval-optimize-empty* r) =
wordinterval-empty (*wordinterval-optimize-empty* r)

by(induction r) fastforce+

private fun *wordinterval-optimize-empty2* **where**
wordinterval-optimize-empty2 (RangeUnion r1 r2) = (let r1o = *wordinterval-optimize-empty* r1;

r2o = *wordinterval-optimize-empty* r2

```

in if
  wordinterval-empty-shallow r1o
then
  r2o
else if
  wordinterval-empty-shallow r2o
then
  r1o
else
  RangeUnion r1o r2o |
wordinterval-optimize-empty2 r = r
lemma wordinterval-optimize-empty-code[code-unfold]:
wordinterval-optimize-empty = wordinterval-optimize-empty2
by (subst fun-eq-iff, clarify, rename-tac r, induct-tac r)
(unfold wordinterval-optimize-empty.simps wordinterval-optimize-empty2.simps
  Let-def helper-optimize-shallow, simp-all)
end

```

Merging overlapping intervals

context
begin

private definition *disjoint* :: 'a set ⇒ 'a set ⇒ bool **where**
disjoint A B ≡ A ∩ B = {}

private primrec *interval-of* :: ('a::len) word × 'a word ⇒ 'a word set **where**
interval-of (s,e) = {s .. e}
declare *interval-of.simps*[simp del]

private definition *disjoint-intervals*
:: (('a::len) word × ('a::len) word) ⇒ ('a word × 'a word) ⇒ bool
where
disjoint-intervals A B ≡ disjoint (interval-of A) (interval-of B)

private definition *not-disjoint-intervals*
:: (('a::len) word × ('a::len) word) ⇒ ('a word × 'a word) ⇒ bool
where
not-disjoint-intervals A B ≡ ¬ disjoint (interval-of A) (interval-of B)

private lemma [code]:
not-disjoint-intervals A B =
(case A of (s,e) ⇒ case B of (s',e') ⇒ s ≤ e' ∧ s' ≤ e ∧ s ≤ e ∧ s' ≤ e')
apply(cases A, cases B)
apply(simp add: not-disjoint-intervals-def interval-of.simps disjoint-def)
done

private lemma [code]:
disjoint-intervals A B =
(case A of (s,e) ⇒ case B of (s',e') ⇒ s > e' ∨ s' > e ∨ s > e ∨ s' > e')

```

apply(cases A, cases B)
apply(simp add: disjoint-intervals-def interval-of.simps disjoint-def)
by fastforce

```

BEGIN merging overlapping intervals

```

private fun merge-overlap
  :: (('a::len) word × ('a::len) word) ⇒ ('a word × 'a word) list ⇒ ('a word ×
'a word) list

```

where

```

merge-overlap s [] = [s] |
merge-overlap (s,e) ((s',e')#ss) = (
  if not-disjoint-intervals (s,e) (s',e')
  then (min s s', max e e')#ss
  else (s',e')#merge-overlap (s,e) ss)

```

private lemma not-disjoint-union:

```

fixes s :: ('a::len) word
shows ¬ disjoint {s..e} {s'..e'} ⇒ {s..e} ∪ {s'..e'} = {min s s' .. max e e'}
by(auto simp add: disjoint-def min-def max-def)

```

private lemma disjoint-subset: disjoint A B ⇒ A ⊆ B ∪ C ⇒ A ⊆ C

```

unfolding disjoint-def
by blast

```

private lemma merge-overlap-helper1: interval-of A ⊆ (∪ s ∈ set ss. interval-of s) ⇒

```

(∪ s ∈ set (merge-overlap A ss). interval-of s) = (∪ s ∈ set ss. interval-of s)
apply(induction ss)
apply(simp; fail)
apply(rename-tac x xs)
apply(cases A, rename-tac a b)
apply(case-tac x)
apply(simp add: not-disjoint-intervals-def interval-of.simps)
apply(intro impI conjI)
apply(drule not-disjoint-union)
apply blast
apply(drule-tac C=(∪ x ∈ set xs. interval-of x) in disjoint-subset)
apply(simp-all)
done

```

private lemma merge-overlap-helper2: ∃ s' ∈ set ss. ¬ disjoint (interval-of A) (interval-of s') ⇒

interval-of A ∪ (∪ s ∈ set ss. interval-of s) = (∪ s ∈ set (merge-overlap A ss). interval-of s)

```

apply(induction ss)
apply(simp; fail)
apply(rename-tac x xs)
apply(cases A, rename-tac a b)
apply(case-tac x)

```

```

apply(simp add: not-disjoint-intervals-def interval-of.simps)
apply(intro impI conjI)
apply(drule not-disjoint-union)
apply blast
apply(simp)
by blast

```

```

private lemma merge-overlap-length:
   $\exists s' \in \text{set } ss. \neg \text{disjoint } (\text{interval-of } A) (\text{interval-of } s') \implies$ 
  length (merge-overlap A ss) = length ss
apply(induction ss)
apply(simp)
apply(rename-tac x xs)
apply(cases A, rename-tac a b)
apply(case-tac x)
apply(simp add: not-disjoint-intervals-def interval-of.simps)
done

```

```

lemma merge-overlap (1:: 16 word,2) [(1, 7)] = [(1, 7)] by eval
lemma merge-overlap (1:: 16 word,2) [(2, 7)] = [(1, 7)] by eval
lemma merge-overlap (1:: 16 word,2) [(3, 7)] = [(3, 7), (1,2)] by eval

```

```

private function listwordinterval-compress
  :: (('a::len) word × ('a::len) word) list  $\Rightarrow$  ('a word × 'a word) list where
  listwordinterval-compress [] = [] |
  listwordinterval-compress (s#ss) = (
    if  $\forall s' \in \text{set } ss. \text{disjoint-intervals } s s'$ 
    then s#listwordinterval-compress ss
    else listwordinterval-compress (merge-overlap s ss))
by(pat-completeness, auto)

```

```

private termination listwordinterval-compress
apply (relation measure length)
apply(rule wf-measure)
apply(simp)
using disjoint-intervals-def merge-overlap-length by fastforce

```

```

private lemma listwordinterval-compress:
   $(\bigcup s \in \text{set } (\text{listwordinterval-compress } ss). \text{interval-of } s) = (\bigcup s \in \text{set } ss. \text{inter-}$ 
   $\text{val-of } s)$ 
apply(induction ss rule: listwordinterval-compress.induct)
apply(simp)
apply(simp)
apply(intro impI)
apply(simp add: disjoint-intervals-def)
apply(drule merge-overlap-helper2)
apply(simp)
done

```

lemma *listwordinterval-compress* $[(1::32 \text{ word}, 3), (8, 10), (2, 5), (3, 7)] = [(8, 10), (1, 7)]$
by *eval*

private lemma *A-in-listwordinterval-compress*: $A \in \text{set } (\text{listwordinterval-compress } ss) \implies$
interval-of $A \subseteq (\bigcup s \in \text{set } ss. \text{interval-of } s)$
using *listwordinterval-compress* **by** *blast*

private lemma *listwordinterval-compress-disjoint*:
 $A \in \text{set } (\text{listwordinterval-compress } ss) \implies B \in \text{set } (\text{listwordinterval-compress } ss) \implies$
 $A \neq B \implies \text{disjoint } (\text{interval-of } A) (\text{interval-of } B)$
apply(*induction* *ss* *arbitrary*: *rule*: *listwordinterval-compress.induct*)
apply(*simp*)
apply(*simp* *split*: *if-split-asm*)
apply(*elim* *disjE*)
apply(*simp-all*)
apply(*simp-all* *add*: *disjoint-intervals-def* *disjoint-def*)
apply(*blast* *dest*: *A-in-listwordinterval-compress*)
done

END merging overlapping intervals

BEGIN merging adjacent intervals

private fun *merge-adjacent*
 $:: ((\text{'a}::\text{len}) \text{ word} \times (\text{'a}::\text{len}) \text{ word}) \Rightarrow (\text{'a} \text{ word} \times \text{'a} \text{ word}) \text{ list} \Rightarrow (\text{'a} \text{ word} \times \text{'a} \text{ word}) \text{ list}$

where

merge-adjacent $s \ [] = [s] \ |$
merge-adjacent $(s, e) ((s', e') \# ss) = ($
if $s \leq e \wedge s' \leq e' \wedge \text{word-next } e = s'$
then $(s, e') \# ss$
else if $s \leq e \wedge s' \leq e' \wedge \text{word-next } e' = s$
then $(s', e) \# ss$
else $(s', e') \# \text{merge-adjacent } (s, e) \text{ } ss)$

private lemma *merge-adjacent-helper*:
interval-of $A \cup (\bigcup s \in \text{set } ss. \text{interval-of } s) = (\bigcup s \in \text{set } (\text{merge-adjacent } A \text{ } ss). \text{interval-of } s)$

apply(*induction* *ss*)
apply(*simp*; *fail*)
apply(*rename-tac* *x* *xs*)
apply(*cases* *A*, *rename-tac* *a* *b*)
apply(*case-tac* *x*)
apply(*simp* *add*: *interval-of.simps*)
apply(*intro* *impI* *conjI*)
apply (*metis* *Un-assoc* *word-adjacent-union*)
apply(*elim* *conjE*)
apply(*drule*(2) *word-adjacent-union*)

subgoal by (*blast*)
subgoal by (*metis word-adjacent-union Un-assoc*)
by *blast*

private lemma *merge-adjacent-length*:

$\exists (s', e') \in \text{set } ss. s \leq e \wedge s' \leq e' \wedge (\text{word-next } e = s' \vee \text{word-next } e' = s)$
 $\implies \text{length } (\text{merge-adjacent } (s, e) ss) = \text{length } ss$

apply(*induction ss*)
apply(*simp*)
apply(*rename-tac x xs*)
apply(*case-tac x*)
apply *simp*
by *blast*

private function *listwordinterval-adjacent*

$:: (('a::\text{len}) \text{ word} \times ('a::\text{len}) \text{ word}) \text{ list} \Rightarrow ('a \text{ word} \times 'a \text{ word}) \text{ list}$ **where**

listwordinterval-adjacent [] = [] |

listwordinterval-adjacent ((*s, e*)#*ss*) = (

$\text{if } \forall (s', e') \in \text{set } ss. \neg (s \leq e \wedge s' \leq e' \wedge (\text{word-next } e = s' \vee \text{word-next } e' = s))$

$\text{then } (s, e) \# \text{listwordinterval-adjacent } ss$

$\text{else } \text{listwordinterval-adjacent } (\text{merge-adjacent } (s, e) ss)$

by(*pat-completeness, auto*)

private termination *listwordinterval-adjacent*

apply (*relation measure length*)

apply(*rule wf-measure*)

apply(*simp*)

apply(*simp*)

using *merge-adjacent-length by fastforce*

private lemma *listwordinterval-adjacent*:

$(\bigcup s \in \text{set } (\text{listwordinterval-adjacent } ss). \text{interval-of } s) = (\bigcup s \in \text{set } ss. \text{interval-of } s)$

apply(*induction ss rule: listwordinterval-adjacent.induct*)

apply(*simp*)

apply(*simp add: merge-adjacent-helper*)

done

lemma *listwordinterval-adjacent* [(1::16 word, 3), (5, 10), (10,10), (4,4)] = [(10, 10), (1, 10)]

by *eval*

END merging adjacent intervals

definition *wordinterval-compress* :: ('a::len) wordinterval \Rightarrow 'a wordinterval

where

wordinterval-compress *r* \equiv

$l2wi (\text{remdups } (\text{listwordinterval-adjacent } (\text{listwordinterval-compress } (\text{wi2l } (\text{wordinterval-optimize-empty } r))))))$

Correctness: Compression preserves semantics

lemma *wordinterval-compress*:

wordinterval-to-set (wordinterval-compress r) = wordinterval-to-set r

unfolding *wordinterval-compress-def*

proof –

have *interval-of'*: *interval-of s = (case s of (s,e) ⇒ {s .. e})* **for** *s*
by (*cases s*) (*simp add: interval-of.simps*)

have *wordinterval-to-set (l2wi (remdups (listwordinterval-adjacent (listwordinterval-compress (wi2l (wordinterval-optimize-empty r)))))) = (∪ x∈set (listwordinterval-adjacent (listwordinterval-compress (wi2l (wordinterval-optimize-empty r))))). interval-of x)*

by (*force simp: interval-of' l2wi*)

also have ... = $(\bigcup s \in \text{set } (wi2l (wordinterval-optimize-empty r)). \text{interval-of } s)$

s)

by (*simp add: listwordinterval-compress listwordinterval-adjacent*)

also have ... = $(\bigcup (i, j) \in \text{set } (wi2l (wordinterval-optimize-empty r)). \{i..j\})$

by (*simp add: interval-of'*)

also have ... = *wordinterval-to-set r* **by** (*simp add: wi2l*)

finally show *wordinterval-to-set*

$(l2wi (remdups (listwordinterval-adjacent (listwordinterval-compress (wi2l (wordinterval-optimize-empty r)))))) = \text{wordinterval-to-set } r .$

qed

end

Example

lemma $(wi2l \circ (\text{wordinterval-compress} :: 32 \text{ wordinterval} \Rightarrow 32 \text{ wordinterval}) \circ l2wi)$

$[(70, 80001), (0,0), (150, 8000), (1,3), (42,41), (3,7), (56, 200), (8,10)]$

=

$[(56, 80001), (0, 10)]$ **by** *eval*

lemma *wordinterval-compress (RangeUnion (RangeUnion (WordInterval (1::32 word) 5)*

(WordInterval 8 10)) (WordInterval 3

7)) =

WordInterval 1 10 **by** *eval*

1.6 Further operations

∪

definition *wordinterval-Union* :: $('a::\text{len}) \text{ wordinterval list} \Rightarrow 'a \text{ wordinterval}$
where

wordinterval-Union ws = wordinterval-compress (foldr wordinterval-union ws Empty-WordInterval)

lemma *wordinterval-Union*:
 $\text{wordinterval-to-set } (\text{wordinterval-Union } ws) = (\bigcup w \in (\text{set } ws). \text{wordinterval-to-set } w)$
by(*induction ws*) (*simp-all add: wordinterval-compress wordinterval-Union-def*)

context

begin

private fun *wordinterval-setminus'*

$:: 'a::\text{len wordinterval} \Rightarrow 'a \text{ wordinterval} \Rightarrow 'a \text{ wordinterval}$ **where**
 $\text{wordinterval-setminus}' (\text{WordInterval } s \ e) (\text{WordInterval } ms \ me) =$
if $s > e \vee ms > me$ *then* $\text{WordInterval } s \ e$ *else*
if $me \geq e$
then
 $\text{WordInterval } (\text{if } ms = 0 \text{ then } 1 \text{ else } s) (\text{min } e \ (\text{word-prev } ms))$
else if $ms \leq s$
then
 $\text{WordInterval } (\text{max } s \ (\text{word-next } me)) (\text{if } me = -1 \text{ then } 0 \text{ else } e)$
else
 $\text{RangeUnion } (\text{WordInterval } (\text{if } ms = 0 \text{ then } 1 \text{ else } s) (\text{word-prev } ms))$
 $(\text{WordInterval } (\text{word-next } me) (\text{if } me = -1 \text{ then } 0 \text{ else } e))$
 $) |$
 $\text{wordinterval-setminus}' (\text{RangeUnion } r1 \ r2) \ t =$
 $\text{RangeUnion } (\text{wordinterval-setminus}' \ r1 \ t) (\text{wordinterval-setminus}' \ r2 \ t) |$
 $\text{wordinterval-setminus}' \ t (\text{RangeUnion } r1 \ r2) =$
 $\text{wordinterval-setminus}' (\text{wordinterval-setminus}' \ t \ r1) \ r2$

private lemma *wordinterval-setminus'-rr-set-eq*:

$\text{wordinterval-to-set}(\text{wordinterval-setminus}' (\text{WordInterval } s \ e) (\text{WordInterval } ms \ me)) =$
 $\text{wordinterval-to-set } (\text{WordInterval } s \ e) - \text{wordinterval-to-set } (\text{WordInterval } ms \ me)$

apply(*simp only: wordinterval-setminus'.simps*)

apply(*case-tac e < s*)

apply *simp*

apply(*case-tac me < ms*)

apply *simp*

apply(*case-tac [!] e ≤ me*)

apply(*case-tac [!] ms = 0*)

apply(*case-tac [!] ms ≤ s*)

apply(*case-tac [!] me = - 1*)

apply(*simp-all add: word-next-unfold word-prev-unfold min-def max-def*)

apply(*safe*)

apply(*auto*)

apply(*uint-arith*)

apply(*uint-arith*)

apply(*uint-arith*)

apply(*uint-arith*)

```

    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    apply(uint-arith)
    done

```

```

private lemma wordinterval-setminus'-set-eq:
  wordinterval-to-set (wordinterval-setminus' r1 r2) =
    wordinterval-to-set r1 - wordinterval-to-set r2
apply (induction rule: wordinterval-setminus'.induct)
  using wordinterval-setminus'-rr-set-eq apply blast
apply auto
done

```

```

lemma wordinterval-setminus'-empty-struct:
  wordinterval-empty r2  $\implies$  wordinterval-setminus' r1 r2 = r1
by (induction r1 r2 rule: wordinterval-setminus'.induct) auto

```

```

definition wordinterval-setminus
  :: 'a::len wordinterval  $\Rightarrow$  'a::len wordinterval  $\Rightarrow$  'a::len wordinterval where
    wordinterval-setminus r1 r2 = wordinterval-compress (wordinterval-setminus'
r1 r2)

```

```

lemma wordinterval-setminus-set-eq[simp]: wordinterval-to-set (wordinterval-setminus
r1 r2) =
  wordinterval-to-set r1 - wordinterval-to-set r2
by (simp add: wordinterval-setminus-def wordinterval-compress wordinterval-setminus'-set-eq)
end

```

```

definition wordinterval-UNIV :: 'a::len wordinterval where
  wordinterval-UNIV  $\equiv$  WordInterval 0 (- 1)

```

```

lemma wordinterval-UNIV-set-eq[simp]: wordinterval-to-set wordinterval-UNIV =
  UNIV
unfolding wordinterval-UNIV-def
using max-word-max by fastforce

```

fun *wordinterval-invert* :: 'a::len wordinterval \Rightarrow 'a::len wordinterval **where**
wordinterval-invert r = *wordinterval-setminus* *wordinterval-UNIV* r
lemma *wordinterval-invert-set-eq*[simp]:
wordinterval-to-set (*wordinterval-invert* r) = *UNIV* - *wordinterval-to-set* r **by**(*auto*)

lemma *wordinterval-invert-UNIV-empty*:
wordinterval-empty (*wordinterval-invert* *wordinterval-UNIV*) **by** *simp*

lemma *wi2l-univ*[simp]: *wi2l* *wordinterval-UNIV* = [(0, - 1)]
unfolding *wordinterval-UNIV-def*
by *simp*

\cap

context

begin

private lemma $\{(s::nat) .. e\} \cap \{s' .. e'\} = \{\}$ \longleftrightarrow $s > e' \vee s' > e \vee s > e \vee s' > e'$

by *simp* *linarith*

private fun *wordinterval-intersection'*

:: 'a::len wordinterval \Rightarrow 'a::len wordinterval \Rightarrow 'a::len wordinterval **where**

wordinterval-intersection' (WordInterval s e) (WordInterval s' e') = (

if $s > e \vee s' > e' \vee s > e' \vee s' > e \vee s > e \vee s' > e'$

then

Empty-WordInterval

else

WordInterval (max s s') (min e e')

) |

wordinterval-intersection' (RangeUnion r1 r2) t =

RangeUnion (*wordinterval-intersection'* r1 t) (*wordinterval-intersection'* r2

t)|

wordinterval-intersection' t (RangeUnion r1 r2) =

RangeUnion (*wordinterval-intersection'* t r1) (*wordinterval-intersection'* t

r2)

private lemma *wordinterval-intersection'-set-eq*:

wordinterval-to-set (*wordinterval-intersection'* r1 r2) =

wordinterval-to-set r1 \cap *wordinterval-to-set* r2

by(*induction* r1 r2 *rule: wordinterval-intersection'.induct*) (*auto*)

lemma *wordinterval-intersection'*

(RangeUnion (RangeUnion (WordInterval (1::32 word) 3) (WordInterval

8 10))

(WordInterval 1 3)) (WordInterval 1 3) =

RangeUnion (RangeUnion (WordInterval 1 3) (WordInterval 1 0))

(WordInterval 1 3) **by** *eval*

definition *wordinterval-intersection*

:: 'a::len wordinterval \Rightarrow 'a::len wordinterval \Rightarrow 'a::len wordinterval **where**

$\text{wordinterval-intersection } r1 \ r2 \equiv \text{wordinterval-compress } (\text{wordinterval-intersection}' \ r1 \ r2)$

lemma *wordinterval-intersection-set-eq*[simp]:
 $\text{wordinterval-to-set } (\text{wordinterval-intersection } r1 \ r2) =$
 $\text{wordinterval-to-set } r1 \cap \text{wordinterval-to-set } r2$
by(simp add: *wordinterval-intersection-def*
 $\text{wordinterval-compress } \text{wordinterval-intersection}'\text{-set-eq}$)

lemma *wordinterval-intersection*
 $(\text{RangeUnion } (\text{RangeUnion } (\text{WordInterval } (1::32 \ \text{word}) \ 3) \ (\text{WordInterval}$
 $8 \ 10)))$
 $(\text{WordInterval } 1 \ 3) \ (\text{WordInterval } 1 \ 3) =$
 $\text{WordInterval } 1 \ 3$ **by** *eval*

end

definition *wordinterval-subset* :: 'a::len wordinterval \Rightarrow 'a::len wordinterval \Rightarrow bool
where

$\text{wordinterval-subset } r1 \ r2 \equiv \text{wordinterval-empty } (\text{wordinterval-setminus } r1 \ r2)$

lemma *wordinterval-subset-set-eq*[simp]:
 $\text{wordinterval-subset } r1 \ r2 = (\text{wordinterval-to-set } r1 \subseteq \text{wordinterval-to-set } r2)$
unfolding *wordinterval-subset-def* **by** *simp*

definition *wordinterval-eq* :: 'a::len wordinterval \Rightarrow 'a::len wordinterval \Rightarrow bool
where

$\text{wordinterval-eq } r1 \ r2 = (\text{wordinterval-subset } r1 \ r2 \wedge \text{wordinterval-subset } r2 \ r1)$

lemma *wordinterval-eq-set-eq*:
 $\text{wordinterval-eq } r1 \ r2 \iff \text{wordinterval-to-set } r1 = \text{wordinterval-to-set } r2$
unfolding *wordinterval-eq-def* **by** *auto*

thm *iffD1*[OF *wordinterval-eq-set-eq*]

lemma *wordinterval-eq-comm*: $\text{wordinterval-eq } r1 \ r2 \iff \text{wordinterval-eq } r2 \ r1$
unfolding *wordinterval-eq-def* **by** *fast*

lemma *wordinterval-to-set-alt*: $\text{wordinterval-to-set } r = \{x. \text{wordinterval-element } x \ r\}$
unfolding *wordinterval-element-set-eq* **by** *blast*

lemma *wordinterval-un-empty*:
 $\text{wordinterval-empty } r1 \implies \text{wordinterval-eq } (\text{wordinterval-union } r1 \ r2) \ r2$
by(subst *wordinterval-eq-set-eq*, *simp*)

lemma *wordinterval-un-empt-b*:
 $\text{wordinterval-empty } r2 \implies \text{wordinterval-eq } (\text{wordinterval-union } r1 \ r2) \ r1$
by(subst *wordinterval-eq-set-eq*, *simp*)

lemma *wordinterval-Diff-triv*:
 $\text{wordinterval-empty } (\text{wordinterval-intersection } a \ b) \implies \text{wordinterval-eq } (\text{wordinterval-setminus}$

a b) *a*
unfolding *wordinterval-eq-set-eq*
by *simp blast*

A size of the datatype, does not correspond to the cardinality of the corresponding set

fun *wordinterval-size* :: ('a::len) *wordinterval* \Rightarrow *nat* **where**
wordinterval-size (*RangeUnion* *a b*) = *wordinterval-size* *a* + *wordinterval-size* *b* |
wordinterval-size (*WordInterval* *s e*) = (if *s* \leq *e* then 1 else 0)

lemma *wordinterval-size-length*: *wordinterval-size* *r* = *length* (*wi2l* *r*)
by(*induction* *r*) (*auto*)

lemma *Ex-wordinterval-nonempty*: $\exists x::('a::len \text{ wordinterval}). y \in \text{wordinterval-to-set } x$

proof show $y \in \text{wordinterval-to-set } \text{wordinterval-UNIV}$ **by** *simp qed*

lemma *wordinterval-eq-reflp*:
reflp *wordinterval-eq*
apply(*rule* *reflpI*)
by(*simp only*: *wordinterval-eq-set-eq*)

lemma *wordinterval-eq-symp*:
symp *wordinterval-eq*
apply(*rule* *sympI*)
by(*simp add*: *wordinterval-eq-comm*)

lemma *wordinterval-eq-transp*:
transp *wordinterval-eq*
apply(*rule* *transpI*)
by(*simp only*: *wordinterval-eq-set-eq*)

lemma *wordinterval-eq-equivp*:
equivp *wordinterval-eq*
by (*auto intro*: *equivpI* *wordinterval-eq-reflp* *wordinterval-eq-symp* *wordinterval-eq-transp*)

The smallest element in the interval

definition *is-lowest-element* :: 'a::ord \Rightarrow 'a *set* \Rightarrow *bool* **where**
is-lowest-element *x S* = ($x \in S \wedge (\forall y \in S. y \leq x \longrightarrow y = x)$)

lemma
fixes *x* :: 'a :: *complete-lattice*
assumes $x \in S$
shows $x = \text{Inf } S \implies \text{is-lowest-element } x S$
using *assms* **apply**(*simp add*: *is-lowest-element-def*)
by (*simp add*: *Inf-lower eq-iff*)

lemma
fixes *x* :: 'a :: *linorder*
assumes *finite S* **and** $x \in S$

```

shows is-lowest-element  $x S \longleftrightarrow x = \text{Min } S$ 
apply(rule)
subgoal
apply(simp add: is-lowest-element-def)
apply(subst Min-eqI[symmetric])
using assms by(auto)
by (metis Min.coboundedI assms(1) assms(2) dual-order.antisym is-lowest-element-def)

```

Smallest element in the interval

```

fun wordinterval-lowest-element :: 'a::len wordinterval  $\Rightarrow$  'a word option where
  wordinterval-lowest-element (WordInterval s e) = (if  $s \leq e$  then Some s else
None) |
  wordinterval-lowest-element (RangeUnion A B) =
  (case (wordinterval-lowest-element A, wordinterval-lowest-element B) of
    (Some a, Some b)  $\Rightarrow$  Some (if  $a < b$  then a else b) |
    (None, Some b)  $\Rightarrow$  Some b |
    (Some a, None)  $\Rightarrow$  Some a |
    (None, None)  $\Rightarrow$  None)

```

```

lemma wordinterval-lowest-none-empty: wordinterval-lowest-element r = None
 $\longleftrightarrow$  wordinterval-empty r
proof(induction r)
case WordInterval thus ?case by simp
next
case RangeUnion thus ?case by fastforce
qed

```

```

lemma wordinterval-lowest-element-correct-A:
  wordinterval-lowest-element r = Some x  $\implies$  is-lowest-element x (wordinterval-to-set
r)
unfolding is-lowest-element-def
apply(induction r arbitrary: x rule: wordinterval-lowest-element.induct)
apply(rename-tac rs re x, case-tac  $rs \leq re$ , auto)[1]
apply(subst(asm) wordinterval-lowest-element.simps(2))
apply(rename-tac A B x)
apply(case-tac wordinterval-lowest-element B)
apply(case-tac[!] wordinterval-lowest-element A)
apply(simp-all add: wordinterval-lowest-none-empty)[3]
apply fastforce
done

```

```

lemma wordinterval-lowest-element-set-eq: assumes  $\neg$  wordinterval-empty r
shows (wordinterval-lowest-element r = Some x) = (is-lowest-element x (wordinterval-to-set
r))

```

```

proof(rule iffI)
assume wordinterval-lowest-element r = Some x

```

```

thus is-lowest-element  $x$  (wordinterval-to-set  $r$ )
using wordinterval-lowest-element-correct-A wordinterval-lowest-none-empty
by simp
next
assume is-lowest-element  $x$  (wordinterval-to-set  $r$ )
with assms show (wordinterval-lowest-element  $r = \text{Some } x$ )
proof(induction  $r$  arbitrary:  $x$  rule: wordinterval-lowest-element.induct)
case 1 thus ?case by(simp add: is-lowest-element-def)
next
case (2  $A B x$ )

have is-lowest-RangeUnion: is-lowest-element  $x$  (wordinterval-to-set  $A \cup$ 
wordinterval-to-set  $B$ )  $\implies$ 
is-lowest-element  $x$  (wordinterval-to-set  $A$ )  $\vee$  is-lowest-element  $x$  (wordinterval-to-set
 $B$ )
by(simp add: is-lowest-element-def)

have wordinterval-lowest-element-RangeUnion:
 $\bigwedge a b A B. \text{wordinterval-lowest-element } A = \text{Some } a \implies$ 
 $\text{wordinterval-lowest-element } B = \text{Some } b \implies$ 
 $\text{wordinterval-lowest-element } (\text{RangeUnion } A B) = \text{Some } (\text{min } a b)$ 
by(auto dest!: wordinterval-lowest-element-correct-A simp add: is-lowest-element-def
min-def)

from 2 show ?case
apply(case-tac wordinterval-lowest-element  $B$ )
apply(case-tac![] wordinterval-lowest-element  $A$ )
apply(auto simp add: is-lowest-element-def)[]
apply(subgoal-tac  $\neg$  wordinterval-empty  $A \wedge \neg$  wordinterval-empty  $B$ )
prefer 2
using arg-cong[where  $f = \text{Not}, \text{OF } \text{wordinterval-lowest-none-empty}$ ] apply
blast
apply(drule(1) wordinterval-lowest-element-RangeUnion)
apply(simp split: option.split-asm add: min-def)
apply(drule is-lowest-RangeUnion)
apply(elim disjE)
apply(simp add: is-lowest-element-def)
apply(clarsimp simp add: wordinterval-lowest-none-empty)

apply(simp add: is-lowest-element-def)
apply(clarsimp simp add: wordinterval-lowest-none-empty)
using wordinterval-lowest-element-correct-A[simplified is-lowest-element-def]
by (metis Un-iff not-le)
qed
qed

```

Cardinality approximation for 'a *wordintervals*

context

```

begin
  lemma card-atLeastAtMost-word: fixes s::('a::len) word shows card {s..e} =
  Suc (unat e) - (unat s)
  apply(cases s > e)
  apply(simp)
  apply(subst(asm) Word.word-less-nat-alt)
  apply simp
  apply(subst upto-enum-set-conv2[symmetric])
  apply(subst List.card-set)
  apply(simp add: remdups-enum-upto)
done

fun wordinterval-card :: ('a::len) wordinterval ⇒ nat where
  wordinterval-card (WordInterval s e) = Suc (unat e) - (unat s) |
  wordinterval-card (RangeUnion a b) = wordinterval-card a + wordinterval-card
b

lemma wordinterval-card: wordinterval-card r ≥ card (wordinterval-to-set r)
proof(induction r)
case WordInterval thus ?case by (simp add: card-atLeastAtMost-word)
next
case (RangeUnion r1 r2)
  have card (wordinterval-to-set r1 ∪ wordinterval-to-set r2) ≤
    card (wordinterval-to-set r1) + card (wordinterval-to-set r2)
  using Finite-Set.card-Un-le by blast
  with RangeUnion show ?case by(simp)
qed

With wordinterval-to-set (wordinterval-compress ?r) = wordinterval-to-set
?r it should be possible to get the exact cardinality

end

end
theory Hs-Compat
imports Main
begin

```

2 Definitions inspired by the Haskell World.

```

definition uncurry :: ('b ⇒ 'c ⇒ 'a) ⇒ 'b × 'c ⇒ 'a
where
  uncurry f a ≡ (case a of (x,y) ⇒ f x y)

lemma uncurry-simp[simp]: uncurry f (a,b) = f a b
by(simp add: uncurry-def)

lemma uncurry-curry-id: uncurry ∘ curry = id curry ∘ uncurry = id
by(simp-all add: fun-eq-iff)

```

lemma *uncurry-split*: $P (\text{uncurry } f \ p) \longleftrightarrow (\forall x1 \ x2. p = (x1, x2) \longrightarrow P (f \ x1 \ x2))$

by(*cases p*) *simp*

lemma *uncurry-split-asm*: $P (\text{uncurry } f \ a) \longleftrightarrow \neg(\exists x \ y. a = (x,y) \wedge \neg P (f \ x \ y))$

by(*simp split: uncurry-split*)

lemmas *uncurry-splits* = *uncurry-split uncurry-split-asm*

lemma *uncurry-case-stmt*: $(\text{case } x \ \text{of } (a, b) \Rightarrow f \ a \ b) = \text{uncurry } f \ x$

by(*cases x, simp*)

end

theory *IP-Address*

imports

Word-Lib. Word-Lemmas

Word-Lib. Word-Syntax

Word-Lib. Reversed-Bit-Lists

Hs-Compat

WordInterval

begin

3 Modelling IP Addresses

An IP address is basically an unsigned integer. We model IP addresses of arbitrary lengths.

We will write *'i word* for IP addresses of length $LENGTH('i)$. We use the convention to write *'i* whenever we mean IP addresses instead of generic words. When we will later have theorems with several polymorphic types in it (e.g. arbitrarily extensible packets), this notation makes it easier to spot that type *'i* is for IP addresses.

The files *IPv4.thy* *IPv6.thy* concrete this for IPv4 and IPv6.

The maximum IP address

definition *max-ip-addr* :: *'i::len word* **where**

$max\text{-ip-addr} \equiv \text{of-nat } ((2^{\wedge}(\text{len-of}(TYPE('i)))) - 1)$

lemma *max-ip-addr-max-word*: $max\text{-ip-addr} = - 1$

by (*simp only: max-ip-addr-def of-nat-mask-eq flip: mask-eq-exp-minus-1*) *simp*

lemma *max-ip-addr-max*: $\forall a. a \leq max\text{-ip-addr}$

by(*simp add: max-ip-addr-max-word*)

lemma *range-0-max-UNIV*: $UNIV = \{0 .. max\text{-ip-addr}\}$

by(*simp add: max-ip-addr-max-word*) *fastforce*

lemma *size* ($x::'i::len \text{ word}$) = $\text{len-of}(TYPE('i))$ **by**(*simp add:word-size*)

3.1 Sets of IP Addresses

```

context
  includes bit-operations-syntax
begin

```

Specifying sets with network masks: 192.168.0.0 255.255.255.0

```

definition ipset-from-netmask :: 'i::len word  $\Rightarrow$  'i::len word  $\Rightarrow$  'i::len word set
where

```

```

  ipset-from-netmask addr netmask  $\equiv$ 
    let
      network-prefix = (addr AND netmask)
    in
      {network-prefix .. network-prefix OR (NOT netmask)}

```

Example (pseudo syntax): *ipset-from-netmask* 192.168.1.129 255.255.255.0
 $= \{192.168.1.0 .. 192.168.1.255\}$

A network mask of all ones (i.e. $- 1$).

```

lemma ipset-from-netmask-minusone:
  ipset-from-netmask ip ( $- 1$ ) = {ip} by (simp add: ipset-from-netmask-def)

```

```

lemma ipset-from-netmask-maxword:
  ipset-from-netmask ip ( $- 1$ ) = {ip} by (simp add: ipset-from-netmask-def)

```

```

lemma ipset-from-netmask-zero:
  ipset-from-netmask ip 0 = UNIV by (auto simp add: ipset-from-netmask-def)

```

Specifying sets in Classless Inter-domain Routing (CIDR) notation: 192.168.0.0/24

```

definition ipset-from-cidr :: 'i::len word  $\Rightarrow$  nat  $\Rightarrow$  'i::len word set where
  ipset-from-cidr addr pflength  $\equiv$ 
    ipset-from-netmask addr ((mask pflength) << (len-of(TYPE('i)) - pflength))

```

Example (pseudo syntax): *ipset-from-cidr* 192.168.1.129 24 = {192.168.1.0
 $.. 192.168.1.255\}$

```

lemma (case ipcidr of (base, len)  $\Rightarrow$  ipset-from-cidr base len) = uncurry ipset-from-cidr
ipcidr
  by(simp add: uncurry-case-stmt)

```

```

lemma ipset-from-cidr-0: ipset-from-cidr ip 0 = UNIV
  by(auto simp add: ipset-from-cidr-def ipset-from-netmask-def Let-def)

```

A prefix length of word size gives back the singleton set with the IP address.

Example: 192.168.1.2/32 = {192.168.1.2}

```

lemma ipset-from-cidr-wordlength:
  fixes ip :: 'i::len word
  shows ipset-from-cidr ip (LENGTH('i)) = {ip}
  by (simp add: ipset-from-cidr-def ipset-from-netmask-def)

```

Alternative definition: Considering words as bit lists:

```

lemma ipset-from-cidr-bl:
  fixes addr :: 'i::len word
  shows ipset-from-cidr addr pflength  $\equiv$ 
    ipset-from-netmask addr (of-bl ((replicate pflength True) @
      (replicate ((len-of(TYPE('i))) - pflength))
False))
  by(simp add: ipset-from-cidr-def mask-bl shiftl-of-bl)

```

```

lemma ipset-from-cidr-alt:
  fixes pre :: 'i::len word
  shows ipset-from-cidr pre len =
    {pre AND (mask len << LENGTH('i) - len)
    ..
    pre OR mask (LENGTH('i) - len)}
  apply(simp add: ipset-from-cidr-def ipset-from-netmask-def Let-def)
  apply(simp add: word-oa-dist)
  apply(simp add: NOT-mask-shifted-lenword)
  done

```

```

lemma ipset-from-cidr-alt2:
  fixes base :: 'i::len word
  shows ipset-from-cidr base len =
    ipset-from-netmask base (NOT (mask (LENGTH('i) - len)))
  apply(simp add: ipset-from-cidr-def)
  using NOT-mask-shifted-lenword by (metis word-not-not)

```

In CIDR notation, we cannot express the empty set.

```

lemma ipset-from-cidr-not-empty: ipset-from-cidr base len  $\neq$  {}
  by(simp add: ipset-from-cidr-alt bitmagic-zeroLast-leq-or1Last)

```

Though we can write 192.168.1.2/24, we say that 192.168.0.0/24 is well-formed.

```

lemma ipset-from-cidr-base-wellformed: fixes base:: 'i::len word
  assumes mask (LENGTH('i) - l) AND base = 0
  shows ipset-from-cidr base l = {base .. base OR mask (LENGTH('i) - l)}
  proof -
    have maskshift-eq-not-mask-generic:
      ((mask l << LENGTH('i) - l) :: 'i::len word) = NOT (mask (LENGTH('i)
- l))
    using NOT-mask-shifted-lenword by (metis word-not-not)
    have *: base AND NOT (mask (LENGTH('i) - l)) = base
    unfolding mask-eq-0-eq-x[symmetric] using assms word-bw-comms(1)[of base]
  by simp
  hence **: base AND NOT (mask (LENGTH('i) - l)) OR mask (LENGTH('i)
- l) =
    base OR mask (LENGTH('i) - l) by simp
  have ipset-from-netmask base (NOT (mask (LENGTH('i) - l))) =
    {base .. base || mask (LENGTH('i) - l)}
  by(simp add: ipset-from-netmask-def Let-def ** *)

```

thus *?thesis* **by**(*simp add: ipset-from-cidr-def maskshift-eq-not-mask-generic*)
qed

lemma *ipset-from-cidr-large-pfxlen*:
fixes *ip:: 'i::len word*
assumes $n \geq \text{LENGTH}('i)$
shows *ipset-from-cidr ip n = {ip}*
proof –
have *obviously: mask (LENGTH('i) – n) = 0* **by** (*simp add: assms*)
show *?thesis*
apply(*subst ipset-from-cidr-base-wellforemd*)
subgoal using *assms* **by** *simp*
by (*simp add: obviously*)
qed

lemma *ipset-from-netmask-base-mask-consume*:
fixes *base :: 'i::len word*
shows *ipset-from-netmask (base AND NOT (mask (LENGTH('i) – m)))*
 $(\text{NOT} (\text{mask} (\text{LENGTH}('i) – m)))$
 $=$
ipset-from-netmask base (NOT (mask (LENGTH('i) – m)))
unfolding *ipset-from-netmask-def* **by**(*simp*)

Another definition of CIDR notation: All IP address which are equal on the first $len – n$ bits

definition *ip-cidr-set :: 'i::len word \Rightarrow nat \Rightarrow 'i word set* **where**
ip-cidr-set i r \equiv
 $\{j . i \text{ AND NOT } (\text{mask} (\text{LENGTH}('i) – r)) = j \text{ AND NOT } (\text{mask} (\text{LENGTH}('i) – r))\}$

The definitions are equal

lemma *ipset-from-cidr-eq-ip-cidr-set*:
fixes *base::'i::len word*
shows *ipset-from-cidr base len = ip-cidr-set base len*
proof –
have *maskshift-eq-not-mask-generic*:
 $((\text{mask } len \ll \text{LENGTH}('a) – len) :: 'a::len \text{ word}) = \text{NOT} (\text{mask} (\text{LENGTH}('a) – len))$
using *NOT-mask-shifted-lenword* **by** (*metis word-not-not*)
have *1: mask (len – m) AND base AND NOT (mask (len – m)) = 0*
for *len m* **and** *base::'i::len word*
by(*simp add: word-bw-lcs*)
have *2: mask (LENGTH('i) – len) AND pfxm-p = 0 \implies*
 $(a \in \text{ipset-from-netmask } pfxm-p (\text{NOT} (\text{mask} (\text{LENGTH}('i) – len))))$
 \longleftrightarrow
 $(pfxm-p = \text{NOT} (\text{mask} (\text{LENGTH}('i) – len)) \text{ AND } a)$ **for** *a::'i::len word*
and *pfxm-p*
apply(*subst ipset-from-cidr-alt2[symmetric]*)

```

apply(subst zero-base-lsb-imp-set-eq-as-bit-operation)
apply(simp; fail)
apply(subst ipset-from-cidr-base-wellforemd)
apply(simp; fail)
apply(simp)
done
from 2[OF 1, of - base] have
  (x ∈ ipset-from-netmask base (~~ (mask (LENGTH('i) - len)))) ↔
  (base && ~~ (mask (LENGTH('i) - len)) = x && ~~ (mask (LENGTH('i)
- len))) for x
apply(simp add: ipset-from-netmask-base-mask-consume)
unfolding word-bw-comms(1)[of - ~~ (mask (LENGTH('i) - len))] by simp
then show ?thesis
  unfolding ip-cidr-set-def ipset-from-cidr-def
  by(auto simp add: maskshift-eq-not-mask-generic)
qed

```

```

lemma ip-cidr-set-change-base: j ∈ ip-cidr-set i r ⇒ ip-cidr-set j r = ip-cidr-set
i r
by (auto simp: ip-cidr-set-def)

```

3.2 IP Addresses as WordIntervals

The nice thing is: *'i wordintervals* are executable.

```

definition iprange-single :: 'i::len word ⇒ 'i wordinterval where
  iprange-single ip ≡ WordInterval ip ip

```

```

fun iprange-interval :: ('i::len word × 'i::len word) ⇒ 'i wordinterval where
  iprange-interval (ip-start, ip-end) = WordInterval ip-start ip-end
declare iprange-interval.simps[simp del]

```

```

lemma iprange-interval-uncurry: iprange-interval ipcidr = uncurry WordInterval
ipcidr
by(cases ipcidr) (simp add: iprange-interval.simps)

```

```

lemma wordinterval-to-set (iprange-single ip) = {ip}
by(simp add: iprange-single-def)

```

```

lemma wordinterval-to-set (iprange-interval (ip1, ip2)) = {ip1 .. ip2}
by(simp add: iprange-interval.simps)

```

Now we can use the set operations on *'i wordintervals*

```

term wordinterval-to-set
term wordinterval-element
term wordinterval-union
term wordinterval-empty
term wordinterval-setminus
term wordinterval-UNIV
term wordinterval-invert
term wordinterval-intersection

```

term *wordinterval-subset*
term *wordinterval-eq*

3.3 IP Addresses in CIDR Notation

We want to convert IP addresses in CIDR notation to intervals. We already have *ipset-from-cidr*, which gives back a non-executable set. We want to convert to something we can store in an *'i wordinterval*.

fun *ipcidr-to-interval-start* :: (*'i::len word* × *nat*) ⇒ *'i::len word* **where**
ipcidr-to-interval-start (*pre, len*) = (
 let netmask = (*mask len*) << (*LENGTH('i)* - *len*);
 network-prefix = (*pre AND netmask*)
 in network-prefix)
fun *ipcidr-to-interval-end* :: (*'i::len word* × *nat*) ⇒ *'i::len word* **where**
ipcidr-to-interval-end (*pre, len*) = (
 let netmask = (*mask len*) << (*LENGTH('i)* - *len*);
 network-prefix = (*pre AND netmask*)
 in network-prefix OR (NOT netmask))
definition *ipcidr-to-interval* :: (*'i::len word* × *nat*) ⇒ (*'i word* × *'i word*) **where**
ipcidr-to-interval cidr ≡ (*ipcidr-to-interval-start cidr, ipcidr-to-interval-end cidr*)

lemma *ipset-from-cidr-ipcidr-to-interval*:
ipset-from-cidr base len =
 {*ipcidr-to-interval-start (base,len) .. ipcidr-to-interval-end (base,len)*}
by(*simp add: Let-def ipcidr-to-interval-def ipset-from-cidr-def ipset-from-netmask-def*)
declare *ipcidr-to-interval-start.simps[simp del]* *ipcidr-to-interval-end.simps[simp del]*

lemma *ipcidr-to-interval*:
ipcidr-to-interval (base, len) = (*s,e*) ⇒ *ipset-from-cidr base len* = {*s .. e*}
by (*simp add: ipcidr-to-interval-def ipset-from-cidr-ipcidr-to-interval*)

definition *ipcidr-tuple-to-wordinterval* :: (*'i::len word* × *nat*) ⇒ *'i wordinterval*
where
ipcidr-tuple-to-wordinterval iprng ≡ *iprange-interval (ipcidr-to-interval iprng)*

lemma *wordinterval-to-set-ipcidr-tuple-to-wordinterval*:
wordinterval-to-set (ipcidr-tuple-to-wordinterval (b, m)) = *ipset-from-cidr b m*
unfolding *ipcidr-tuple-to-wordinterval-def ipset-from-cidr-ipcidr-to-interval*
 ipcidr-to-interval-def
by(*simp add: iprange-interval.simps*)

lemma *wordinterval-to-set-ipcidr-tuple-to-wordinterval-uncurry*:
wordinterval-to-set (ipcidr-tuple-to-wordinterval ipcidr) = *uncurry ipset-from-cidr ipcidr*
by(*cases ipcidr, simp add: wordinterval-to-set-ipcidr-tuple-to-wordinterval*)

definition *ipcidr-union-set* :: ('i::len word × nat) set ⇒ ('i word) set **where**
ipcidr-union-set ips ≡ $\bigcup (base, len) \in ips. ipset-from-cidr\ base\ len$

lemma *ipcidr-union-set-uncurry*:

ipcidr-union-set ips = $(\bigcup ipcidr \in ips. uncurry\ ipset-from-cidr\ ipcidr)$
by(*simp add: ipcidr-union-set-def uncurry-case-stmt*)

3.4 Clever Operations on IP Addresses in CIDR Notation

Intersecting two intervals may result in a new interval. Example: $\{1..10\} \cap \{5..20\} = \{5..10\}$

Intersecting two IP address ranges represented as CIDR ranges results either in the empty set or the smaller of the two ranges. It will never create a new range.

context
begin

private lemma *less-and-not-mask-eq*:

fixes *i* :: ('a :: len) word
assumes $r2 \leq r1$ *i* && $\sim\sim (mask\ r2) = x$ && $\sim\sim (mask\ r2)$
shows *i* && $\sim\sim (mask\ r1) = x$ && $\sim\sim (mask\ r1)$

proof –

have *i* AND NOT (*mask* *r1*) = (*i* && $\sim\sim (mask\ r2)$) && $\sim\sim (mask\ r1)$ (**is**
 - = ?*w* && -)

using $\langle r2 \leq r1 \rangle$ **by** (*simp add: and-not-mask-twice max-def*)

also have ?*w* = *x* && $\sim\sim (mask\ r2)$ **by** *fact*

also have ... && $\sim\sim (mask\ r1) = x$ && $\sim\sim (mask\ r1)$

using $\langle r2 \leq r1 \rangle$ **by** (*simp add: and-not-mask-twice max-def*)

finally show ?*thesis* .

qed

lemma *ip-cidr-set-less*:

fixes *i* :: 'i::len word

shows $r1 \leq r2 \implies ip-cidr-set\ i\ r2 \subseteq ip-cidr-set\ i\ r1$

unfolding *ip-cidr-set-def*

apply *auto*

apply (*rule less-and-not-mask-eq*[**where** ?*r2*.0=LENGTH('i) – *r2*])

apply *auto*

done

private lemma *ip-cidr-set-intersect-subset-helper*:

fixes *i1 r1 i2 r2*

assumes *disj*: $ip-cidr-set\ i1\ r1 \cap ip-cidr-set\ i2\ r2 \neq \{\}$ **and** $r1 \leq r2$

shows $ip-cidr-set\ i2\ r2 \subseteq ip-cidr-set\ i1\ r1$

proof –

from *disj* **obtain** *j* **where** $j \in ip-cidr-set\ i1\ r1$ $j \in ip-cidr-set\ i2\ r2$ **by** *auto*

with $\langle r1 \leq r2 \rangle$ **have** $j \in ip-cidr-set\ j\ r1$ $j \in ip-cidr-set\ j\ r1$

using *ip-cidr-set-change-base ip-cidr-set-less* **by** *blast+*

```

    show ip-cidr-set i2 r2  $\subseteq$  ip-cidr-set i1 r1
  proof
    fix i assume i  $\in$  ip-cidr-set i2 r2
    with  $\langle j \in \text{ip-cidr-set } i2 \text{ } r2 \rangle$  have i  $\in$  ip-cidr-set j r2 using ip-cidr-set-change-base
  by auto
    also have ip-cidr-set j r2  $\subseteq$  ip-cidr-set j r1 using  $\langle r1 \leq r2 \rangle$  ip-cidr-set-less
  by blast
    also have ... = ip-cidr-set i1 r1 using  $\langle j \in \text{ip-cidr-set } i1 \text{ } r1 \rangle$  ip-cidr-set-change-base
  by blast
    finally show i  $\in$  ip-cidr-set i1 r1 .
  qed
end

```

```

lemma ip-cidr-set-notsubset-empty-inter:
   $\neg$  ip-cidr-set i1 r1  $\subseteq$  ip-cidr-set i2 r2  $\implies$ 
   $\neg$  ip-cidr-set i2 r2  $\subseteq$  ip-cidr-set i1 r1  $\implies$ 
  ip-cidr-set i1 r1  $\cap$  ip-cidr-set i2 r2 = {}
  apply(cases r1  $\leq$  r2)
  subgoal using ip-cidr-set-intersect-subset-helper by blast
  apply(cases r2  $\leq$  r1)
  subgoal using ip-cidr-set-intersect-subset-helper by blast
  by(simp)
end

```

```

lemma ip-cidr-intersect:
   $\neg$  ipset-from-cidr b2 m2  $\subseteq$  ipset-from-cidr b1 m1  $\implies$ 
   $\neg$  ipset-from-cidr b1 m1  $\subseteq$  ipset-from-cidr b2 m2  $\implies$ 
  ipset-from-cidr b1 m1  $\cap$  ipset-from-cidr b2 m2 = {}
  apply(simp add: ipset-from-cidr-eq-ip-cidr-set)
  using ip-cidr-set-notsubset-empty-inter by blast

```

Computing the intersection of two IP address ranges in CIDR notation

```

fun ipcidr-conjunct :: ('i::len word  $\times$  nat)  $\Rightarrow$  ('i word  $\times$  nat)  $\Rightarrow$  ('i word  $\times$  nat)
option where
  ipcidr-conjunct (base1, m1) (base2, m2) = (
    if
      ipset-from-cidr base1 m1  $\cap$  ipset-from-cidr base2 m2 = {}
    then
      None
    else if
      ipset-from-cidr base1 m1  $\subseteq$  ipset-from-cidr base2 m2
    then
      Some (base1, m1)
    else
      Some (base2, m2)
  )

```

Intersecting with an address with prefix length zero always yields a non-

empty result.

lemma *ipcidr-conjunct-any*: *ipcidr-conjunct a (x,0) ≠ None ipcidr-conjunct (y,0)*
b ≠ None

apply(*cases a, simp add: ipset-from-cidr-0 ipset-from-cidr-not-empty*)
by(*cases b, simp add: ipset-from-cidr-0 ipset-from-cidr-not-empty*)

lemma *ipcidr-conjunct-correct*: (*case ipcidr-conjunct (b1, m1) (b2, m2)*
of Some (bx, mx) ⇒ ipset-from-cidr bx mx
| None ⇒ {}) =
(ipset-from-cidr b1 m1) ∩ (ipset-from-cidr b2 m2)

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

using *ip-cidr-intersect* **by** *fast*

declare *ipcidr-conjunct.simps[simp del]*

3.5 Code Equations

Executable definition using word intervals

lemma *ipcidr-conjunct-word[code]*:

ipcidr-conjunct ips1 ips2 = (
if
wordinterval-empty (wordinterval-intersection
(ipcidr-tuple-to-wordinterval ips1) (ipcidr-tuple-to-wordinterval
ips2))
then
None
else if
wordinterval-subset (ipcidr-tuple-to-wordinterval ips1) (ipcidr-tuple-to-wordinterval
ips2)
then
Some ips1
else
Some ips2
)
apply(*simp*)
apply(*cases ips1, cases ips2, rename-tac b1 m1 b2 m2, simp*)
apply(*auto simp add: wordinterval-to-set-ipcidr-tuple-to-wordinterval ipcidr-conjunct.simps*
split: if-split-asm)
done

lemma *ipcidr-conjunct (0::32 word,0) (8,1) = Some (8, 1)* **by** *eval*

export-code *ipcidr-conjunct checking SML*

making element check executable

lemma *addr-in-ipset-from-netmask-code[code-unfold]*:

addr ∈ (ipset-from-netmask base netmask) ↔
(base AND netmask) ≤ addr ∧ addr ≤ (base AND netmask) OR (NOT
netmask)

```

    by(simp add: ipset-from-netmask-def Let-def)
lemma addr-in-ipset-from-cidr-code[code-unfold]:
  (addr::'i::len word) ∈ (ipset-from-cidr pre len) ↔
    (pre AND ((mask len) << (LENGTH('i) - len))) ≤ addr ∧
    addr ≤ pre OR (mask (LENGTH('i) - len))
unfolding ipset-from-cidr-alt by simp

```

end

```

theory IPv4
imports IP-Address
        NumberWang-IPv4

```

begin

4 IPv4 Addresses

An IPv4 address is basically a 32 bit unsigned integer.

```

type-synonym ipv4addr = 32 word

```

```

lemma ipv4addr-and-mask-eq-self [simp]:
  ⟨a && 4294967295 = a⟩ for a :: ipv4addr
proof -
  have ⟨take-bit 32 a = a⟩
  by (rule take-bit-word-eq-self) simp
  then show ?thesis
  by (simp add: take-bit-eq-mask mask-numeral)
qed

```

Conversion between natural numbers and IPv4 addresses

```

definition nat-of-ipv4addr :: ipv4addr ⇒ nat where
  nat-of-ipv4addr a = unat a
definition ipv4addr-of-nat :: nat ⇒ ipv4addr where
  ipv4addr-of-nat n = of-nat n

```

The maximum IPv4 address

```

definition max-ipv4-addr :: ipv4addr where
  max-ipv4-addr ≡ ipv4addr-of-nat ((232) - 1)

lemma max-ipv4-addr-number: max-ipv4-addr = 4294967295
  unfolding max-ipv4-addr-def ipv4addr-of-nat-def by(simp)
lemma max-ipv4-addr = 0b11111111111111111111111111111111
  by(fact max-ipv4-addr-number)
lemma max-ipv4-addr-max-word: max-ipv4-addr = - 1
  by(simp add: max-ipv4-addr-number)
lemma max-ipv4-addr-max[simp]: ∀ a. a ≤ max-ipv4-addr

```

```

  by(simp add: max-ipv4-addr-max-word)
lemma UNIV-ipv4addrset: UNIV = {0 .. max-ipv4-addr}
  by(simp add: max-ipv4-addr-max-word) fastforce

```

identity functions

```

lemma nat-of-ipv4addr-ipv4addr-of-nat-mod: nat-of-ipv4addr (ipv4addr-of-nat n)
= n mod 232
  by (simp add: ipv4addr-of-nat-def nat-of-ipv4addr-def unat-of-nat take-bit-eq-mod)
lemma nat-of-ipv4addr-ipv4addr-of-nat:
   $\llbracket n \leq \text{nat-of-ipv4addr max-ipv4-addr} \rrbracket \implies \text{nat-of-ipv4addr (ipv4addr-of-nat } n) = n$ 
  by (simp add: nat-of-ipv4addr-ipv4addr-of-nat-mod max-ipv4-addr-def)
lemma ipv4addr-of-nat-nat-of-ipv4addr: ipv4addr-of-nat (nat-of-ipv4addr addr)
= addr
  by(simp add: ipv4addr-of-nat-def nat-of-ipv4addr-def)

```

4.1 Representing IPv4 Adresses (Syntax)

context

includes bit-operations-syntax

begin

```

fun ipv4addr-of-dotdecimal :: nat × nat × nat × nat ⇒ ipv4addr where
  ipv4addr-of-dotdecimal (a,b,c,d) = ipv4addr-of-nat (d + 256 * c + 65536 * b
+ 16777216 * a )

```

```

fun dotdecimal-of-ipv4addr :: ipv4addr ⇒ nat × nat × nat × nat where
  dotdecimal-of-ipv4addr a = (nat-of-ipv4addr ((a >> 24) AND 0xFF),
  nat-of-ipv4addr ((a >> 16) AND 0xFF),
  nat-of-ipv4addr ((a >> 8) AND 0xFF),
  nat-of-ipv4addr (a AND 0xff))

```

```

declare ipv4addr-of-dotdecimal.simps[simp del]
declare dotdecimal-of-ipv4addr.simps[simp del]

```

Examples:

```

lemma ipv4addr-of-dotdecimal (192, 168, 0, 1) = 3232235521
  by(simp add: ipv4addr-of-dotdecimal.simps ipv4addr-of-nat-def)

```

```

lemma dotdecimal-of-ipv4addr 3232235521 = (192, 168, 0, 1)
  by(simp add: dotdecimal-of-ipv4addr.simps nat-of-ipv4addr-def)

```

a different notation for *ipv4addr-of-dotdecimal*

```

lemma ipv4addr-of-dotdecimal-bit:
  ipv4addr-of-dotdecimal (a,b,c,d) =
  (ipv4addr-of-nat a << 24) + (ipv4addr-of-nat b << 16) +
  (ipv4addr-of-nat c << 8) + ipv4addr-of-nat d
proof -
  have a: (ipv4addr-of-nat a) << 24 = ipv4addr-of-nat (a * 16777216)

```

```

    by(simp add: ipv4addr-of-nat-def shiftl-t2n)
  have b: (ipv4addr-of-nat b) << 16 = ipv4addr-of-nat (b * 65536)
    by(simp add: ipv4addr-of-nat-def shiftl-t2n)
  have c: (ipv4addr-of-nat c) << 8 = ipv4addr-of-nat (c * 256)
    by(simp add: ipv4addr-of-nat-def shiftl-t2n)
  have ipv4addr-of-nat-suc:  $\bigwedge x. \text{ipv4addr-of-nat} (\text{Suc } x) = \text{word-succ} (\text{ipv4addr-of-nat } x)$ 
  by(simp add: ipv4addr-of-nat-def, metis Abs-fnat-hom-Suc of-nat-Suc)
} from this a b c
show ?thesis
  apply(simp add: ipv4addr-of-dotdecimal.simps)
  apply(rule arg-cong[where f=ipv4addr-of-nat])
  apply(thin-tac -)+
  by presburger
qed

```

lemma *size-ipv4addr*: $\text{size } (x::\text{ipv4addr}) = 32$ **by**(simp add:word-size)

lemma *dotdecimal-of-ipv4addr-ipv4addr-of-dotdecimal*:

$\llbracket a < 256; b < 256; c < 256; d < 256 \rrbracket \implies$

$\text{dotdecimal-of-ipv4addr} (\text{ipv4addr-of-dotdecimal } (a,b,c,d)) = (a,b,c,d)$

proof –

assume $a < 256$ **and** $b < 256$ **and** $c < 256$ **and** $d < 256$

note *assms* = $\langle a < 256 \rangle \langle b < 256 \rangle \langle c < 256 \rangle \langle d < 256 \rangle$

hence $a: \text{nat-of-ipv4addr} ((\text{ipv4addr-of-nat } (d + 256 * c + 65536 * b + 16777216 * a) \ggg 24) \text{ AND mask } 8) = a$

apply (simp only: flip: take-bit-eq-mask)

apply (simp add: ipv4addr-of-nat-def nat-of-ipv4addr-def)

apply transfer

apply (simp add: drop-bit-take-bit nat-take-bit-eq flip: take-bit-eq-mask)

apply (simp add: drop-bit-eq-div take-bit-eq-mod)

done

have *ipv4addr-of-nat-AND-mask8*: $(\text{ipv4addr-of-nat } a) \text{ AND mask } 8 = (\text{ipv4addr-of-nat } (a \bmod 256))$

for a

apply (simp only: flip: take-bit-eq-mask)

apply (simp add: ipv4addr-of-nat-def)

apply transfer

apply (simp flip: take-bit-eq-mask)

apply (simp add: take-bit-eq-mod of-nat-mod)

done

from *assms* **have** b :

$\text{nat-of-ipv4addr} ((\text{ipv4addr-of-nat } (d + 256 * c + 65536 * b + 16777216 * a) \ggg 16) \text{ AND mask } 8) = b$

```

apply (simp only: flip: take-bit-eq-mask)
apply (simp add: ipv4addr-of-nat-def nat-of-ipv4addr-def)
apply transfer
apply (simp add: drop-bit-take-bit flip: take-bit-eq-mask)
using div65536
apply (simp add: drop-bit-eq-div take-bit-eq-mod)
done
from assms have c:
  nat-of-ipv4addr ((ipv4addr-of-nat (d + 256 * c + 65536 * b + 16777216 *
a) >> 8) AND mask 8) = c
  apply (simp only: flip: take-bit-eq-mask)
  apply (simp add: ipv4addr-of-nat-def nat-of-ipv4addr-def)
  apply transfer
  apply (simp add: drop-bit-take-bit flip: take-bit-eq-mask)
  using div256
  apply (simp add: drop-bit-eq-div take-bit-eq-mod)
  done
  from <d < 256> have d: nat-of-ipv4addr (ipv4addr-of-nat (d + 256 * c +
65536 * b + 16777216 * a) AND mask 8) = d
  apply (simp only: flip: take-bit-eq-mask)
  apply (simp add: ipv4addr-of-nat-AND-mask8 ipv4addr-of-nat-def nat-of-ipv4addr-def)
  apply transfer
  apply (simp flip: take-bit-eq-mask)
  apply (simp add: take-bit-eq-mod nat-mod-distrib nat-add-distrib nat-mult-distrib
mod256)
  done
  from a b c d show ?thesis
  apply (simp add: ipv4addr-of-dotdecimal.simps dotdecimal-of-ipv4addr.simps
mask-numeral)
  done
qed

```

```

lemma ipv4addr-of-dotdecimal-dotdecimal-of-ipv4addr:
  (ipv4addr-of-dotdecimal (dotdecimal-of-ipv4addr ip)) = ip
proof –
  have ip-and-mask8-bl-drop24: (ip::ipv4addr) AND mask 8 = of-bl (drop 24
(to-bl ip))
  by(simp add: of-drop-to-bl size-ipv4addr)
  have List-rev-drop-geqn: length x ≥ n ⇒ (take n (rev x)) = rev (drop (length
x – n) x)
  for x :: 'a list and n by(simp add: List.rev-drop)
  have and-mask-bl-take: length x ≥ n ⇒ ((of-bl x) AND mask n) = (of-bl (rev
(take n (rev (x)))))
  for x n by(simp add: List-rev-drop-geqn of-bl-drop)
  have ipv4addr-and-255: x AND 255 = take-bit 8 x for x :: ipv4addr
  by (simp add: take-bit-eq-mask mask-numeral)
  have bit-equality:
    ((ip >> 24) AND 0xFF << 24) + ((ip >> 16) AND 0xFF << 16) + ((ip
>> 8) AND 0xFF << 8) + (ip AND 0xFF) =

```

```

    of-bl (take 8 (to-bl ip) @ take 8 (drop 8 (to-bl ip)) @ take 8 (drop 16 (to-bl
ip)) @ drop 24 (to-bl ip))
    apply (simp add: ipv4addr-and-255 shiftr-def shiftr-def rev-drop rev-take
drop-take)
    apply (simp only: of-bl-append mult.commute [of ⟨2 ^ n⟩ for n] flip: push-bit-eq-mult)
    apply (simp add: of-bl-drop-eq-take-bit take-drop of-bl-take-to-bl-eq-drop-bit
take-bit-drop-bit take-bit-word-eq-self)
    done
    have blip-split:  $\bigwedge$  blip. length blip = 32  $\implies$ 
      blip = (take 8 blip) @ (take 8 (drop 8 blip)) @ (take 8 (drop 16 blip)) @ (take
8 (drop 24 blip))
    by(rename-tac blip,case-tac blip,simp-all)+
    have ipv4addr-of-dotdecimal (dotdecimal-of-ipv4addr ip) = of-bl (to-bl ip)
    apply (subst blip-split)
    apply simp
    apply (simp add: ipv4addr-of-dotdecimal-bit dotdecimal-of-ipv4addr.simps)
    apply (simp add: ipv4addr-of-nat-nat-of-ipv4addr)
    apply (simp flip: bit-equality)
    done
    thus ?thesis using word-bl.Rep-inverse[symmetric] by simp
qed

```

```

lemma ipv4addr-of-dotdecimal-eqE:
   $\llbracket$  ipv4addr-of-dotdecimal (a,b,c,d) = ipv4addr-of-dotdecimal (e,f,g,h);
  a < 256; b < 256; c < 256; d < 256; e < 256; f < 256; g < 256; h < 256
 $\rrbracket \implies$ 
  a = e  $\wedge$  b = f  $\wedge$  c = g  $\wedge$  d = h
  by (metis Pair-inject dotdecimal-of-ipv4addr-ipv4addr-of-dotdecimal)

```

4.2 IP Ranges: Examples

```

lemma (UNIV :: ipv4addr set) = {0 .. max-ipv4-addr} by (simp add: UNIV-ipv4addrset)
lemma (42::ipv4addr)  $\in$  UNIV by (simp)

```

```

lemma ipset-from-netmask (ipv4addr-of-dotdecimal (192,168,0,42)) (ipv4addr-of-dotdecimal
(255,255,0,0)) =
  {ipv4addr-of-dotdecimal (192,168,0,0) .. ipv4addr-of-dotdecimal (192,168,255,255)}
  by (simp add: ipset-from-netmask-def ipv4addr-of-dotdecimal.simps ipv4addr-of-nat-def)

```

```

lemma ipset-from-netmask (ipv4addr-of-dotdecimal (192,168,0,42)) (ipv4addr-of-dotdecimal
(0,0,0,0)) = UNIV
  by (simp add: UNIV-ipv4addrset ipset-from-netmask-def ipv4addr-of-dotdecimal.simps
ipv4addr-of-nat-def max-ipv4-addr-max-word)

```

192.168.0.0/24

```

lemma fixes addr :: ipv4addr
  shows ipset-from-cidr addr pflength =

```

$ipset\text{-from-netmask } addr \ ((mask \ pflength) \ll (32 - pflength))$
by(*simp add: ipset-from-cidr-def*)

lemma $ipset\text{-from-cidr } (ipv4addr\text{-of-dotdecimal } (192,168,0,42)) \ 16 =$
 $\{ipv4addr\text{-of-dotdecimal } (192,168,0,0) .. ipv4addr\text{-of-dotdecimal } (192,168,255,255)\}$
by(*simp add: ipset-from-cidr-alt mask-eq ipv4addr-of-dotdecimal.simps ipv4addr-of-nat-def*)

lemma $ip \in (ipset\text{-from-cidr } (ipv4addr\text{-of-dotdecimal } (0, 0, 0, 0)) \ 0)$
by(*simp add: ipset-from-cidr-0*)

lemma $ipv4set\text{-from-cidr-32: fixes } addr :: ipv4addr$
shows $ipset\text{-from-cidr } addr \ 32 = \{addr\}$
by (*simp add: ipset-from-cidr-alt mask-numeral*)

lemma **fixes** $pre :: ipv4addr$
shows $ipset\text{-from-cidr } pre \ len = \{(pre \ AND \ ((mask \ len) \ll (32 - len))) ..$
 $pre \ OR \ (mask \ (32 - len))\}$
by (*simp add: ipset-from-cidr-alt ipset-from-cidr-def*)

making element check executable

lemma $addr\text{-in-ipv4set-from-netmask-code}[code-unfold]:$
fixes $addr :: ipv4addr$
shows $addr \in (ipset\text{-from-netmask } base \ netmask) \longleftrightarrow$
 $(base \ AND \ netmask) \leq addr \wedge addr \leq (base \ AND \ netmask) \ OR \ (NOT$
 $netmask)$

by (*simp add: addr-in-ipset-from-netmask-code*)
lemma $addr\text{-in-ipv4set-from-cidr-code}[code-unfold]:$
fixes $addr :: ipv4addr$
shows $addr \in (ipset\text{-from-cidr } pre \ len) \longleftrightarrow$
 $(pre \ AND \ ((mask \ len) \ll (32 - len))) \leq addr \wedge addr \leq pre \ OR$
 $(mask \ (32 - len))$
by(*simp add: addr-in-ipset-from-cidr-code*)

lemma $ipv4addr\text{-of-dotdecimal } (192,168,42,8) \in (ipset\text{-from-cidr } (ipv4addr\text{-of-dotdecimal } (192,168,0,0)) \ 16)$
by (*simp add: ipv4addr-of-nat-def ipset-from-cidr-def ipv4addr-of-dotdecimal.simps ipset-from-netmask-def mask-eq-exp-minus-1 word-le-def*)

definition $ipv4range\text{-UNIV} :: 32 \ wordinterval$ **where** $ipv4range\text{-UNIV} \equiv wordinterval\text{-UNIV}$

lemma $ipv4range\text{-UNIV-set-eq: wordinterval-to-set } ipv4range\text{-UNIV} = UNIV$
by(*simp only: ipv4range-UNIV-def wordinterval-UNIV-set-eq*)

thm $iffD1[OF \ wordinterval\text{-eq-set-eq}]$

This $LENGTH('a)$ is 32 for IPv4 addresses.

```

lemma ipv4cidr-to-interval-simps[code-unfold]: ipcidr-to-interval ((pre::ipv4addr),
len) = (
  let netmask = (mask len) << (32 - len);
    network-prefix = (pre AND netmask)
  in (network-prefix, network-prefix OR (NOT netmask)))
by(simp add: ipcidr-to-interval-def Let-def ipcidr-to-interval-start.simps ipcidr-to-interval-end.simps)

end

end

```

```

theory IPv6
imports
  IP-Address
  NumberWang-IPv6

```

```

begin

```

5 IPv6 Addresses

An IPv6 address is basically a 128 bit unsigned integer. RFC 4291, Section 2.

```

type-synonym ipv6addr = 128 word

```

Conversion between natural numbers and IPv6 addresses

```

definition nat-of-ipv6addr :: ipv6addr  $\Rightarrow$  nat where
  nat-of-ipv6addr a = unat a
definition ipv6addr-of-nat :: nat  $\Rightarrow$  ipv6addr where
  ipv6addr-of-nat n = of-nat n

```

```

lemma ipv6addr-of-nat n = word-of-int (int n)
by(simp add: ipv6addr-of-nat-def)

```

The maximum IPv6 address

```

definition max-ipv6-addr :: ipv6addr where
  max-ipv6-addr  $\equiv$  ipv6addr-of-nat ((2128) - 1)

```

```

lemma max-ipv6-addr-number: max-ipv6-addr = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
unfolding max-ipv6-addr-def ipv6addr-of-nat-def by(simp)
lemma max-ipv6-addr = 340282366920938463463374607431768211455
by(fact max-ipv6-addr-number)
lemma max-ipv6-addr-max-word: max-ipv6-addr = - 1
by(simp add: max-ipv6-addr-number)
lemma max-ipv6-addr-max:  $\forall a. a \leq \text{max-ipv6-addr}$ 
by(simp add: max-ipv6-addr-max-word)
lemma UNIV-ipv6addrset: UNIV =  $\{0 .. \text{max-ipv6-addr}\}$ 
by(simp add: max-ipv6-addr-max-word) fastforce

```

identity functions

lemma *nat-of-ipv6addr-ipv6addr-of-nat-mod*: *nat-of-ipv6addr (ipv6addr-of-nat n)*
= *n mod 2¹²⁸*
by (*simp add: ipv6addr-of-nat-def nat-of-ipv6addr-def unat-of-nat take-bit-eq-mod*)
lemma *nat-of-ipv6addr-ipv6addr-of-nat*:
n ≤ nat-of-ipv6addr max-ipv6-addr ⇒ nat-of-ipv6addr (ipv6addr-of-nat n) =
n
by (*simp add: nat-of-ipv6addr-ipv6addr-of-nat-mod max-ipv6-addr-def*)
lemma *ipv6addr-of-nat-nat-of-ipv6addr*: *ipv6addr-of-nat (nat-of-ipv6addr addr)*
= *addr*
by(*simp add: ipv6addr-of-nat-def nat-of-ipv6addr-def*)

5.1 Syntax of IPv6 Addresses

RFC 4291, Section 2.2.: Text Representation of Addresses

Quoting the RFC (note: errata exists):

1. The preferred form is x:x:x:x:x:x:x, where the 'x's are one to four hexadecimal digits of the eight 16-bit pieces of the address.

Examples:

ABCD:EF01:2345:6789:ABCD:EF01:2345:6789
2001:DB8:0:0:8:800:200C:417A

datatype *ipv6addr-syntax* =
IPv6AddrPreferred 16 word 16 word 16 word 16 word 16 word 16 word 16 word 16 word
16 word

2. [...] In order to make writing addresses containing zero bits easier, a special syntax is available to compress the zeros. The use of ":" indicates one or more groups of 16 bits of zeros. The "::" can only appear once in an address. The ":::" can also be used to compress leading or trailing zeros in an address.

For example, the following addresses

2001:DB8:0:0:8:800:200C:417A	a unicast address
FF01:0:0:0:0:0:0:101	a multicast address
0:0:0:0:0:0:0:1	the loopback address
0:0:0:0:0:0:0:0	the unspecified address

may be represented as

2001:DB8::8:800:200C:417A	a unicast address
FF01::101	a multicast address
::1	the loopback address
::	the unspecified address

datatype *ipv6addr-syntax-compressed* =

— using *unit* for the omission ::.

Naming convention of the datatype: The first number is the position where the omission occurs. The second number is the length of the specified address pieces. I.e. ‘8 minus the second number’ pieces are omitted.

IPv6AddrCompressed1-0 unit
| *IPv6AddrCompressed1-1 unit 16 word*
| *IPv6AddrCompressed1-2 unit 16 word 16 word*
| *IPv6AddrCompressed1-3 unit 16 word 16 word 16 word*
| *IPv6AddrCompressed1-4 unit 16 word 16 word 16 word 16 word*
| *IPv6AddrCompressed1-5 unit 16 word 16 word 16 word 16 word 16 word*
| *IPv6AddrCompressed1-6 unit 16 word 16 word 16 word 16 word 16 word 16 word*
| *IPv6AddrCompressed1-7 unit 16 word 16 word 16 word 16 word 16 word 16 word 16 word*
16 word

| *IPv6AddrCompressed2-1 16 word unit*
| *IPv6AddrCompressed2-2 16 word unit 16 word*
| *IPv6AddrCompressed2-3 16 word unit 16 word 16 word*
| *IPv6AddrCompressed2-4 16 word unit 16 word 16 word 16 word*
| *IPv6AddrCompressed2-5 16 word unit 16 word 16 word 16 word 16 word*
| *IPv6AddrCompressed2-6 16 word unit 16 word 16 word 16 word 16 word 16 word*
| *IPv6AddrCompressed2-7 16 word unit 16 word 16 word 16 word 16 word 16 word 16 word*
16 word

| *IPv6AddrCompressed3-2 16 word 16 word unit*
| *IPv6AddrCompressed3-3 16 word 16 word unit 16 word*
| *IPv6AddrCompressed3-4 16 word 16 word unit 16 word 16 word*
| *IPv6AddrCompressed3-5 16 word 16 word unit 16 word 16 word 16 word*
| *IPv6AddrCompressed3-6 16 word 16 word unit 16 word 16 word 16 word 16 word*
| *IPv6AddrCompressed3-7 16 word 16 word unit 16 word 16 word 16 word 16 word 16 word*
16 word

| *IPv6AddrCompressed4-3 16 word 16 word 16 word unit*
| *IPv6AddrCompressed4-4 16 word 16 word 16 word unit 16 word*
| *IPv6AddrCompressed4-5 16 word 16 word 16 word unit 16 word 16 word*
| *IPv6AddrCompressed4-6 16 word 16 word 16 word unit 16 word 16 word 16 word*
| *IPv6AddrCompressed4-7 16 word 16 word 16 word unit 16 word 16 word 16 word 16 word*
16 word

| *IPv6AddrCompressed5-4 16 word 16 word 16 word 16 word unit*
| *IPv6AddrCompressed5-5 16 word 16 word 16 word 16 word unit 16 word*
| *IPv6AddrCompressed5-6 16 word 16 word 16 word 16 word unit 16 word 16 word*
| *IPv6AddrCompressed5-7 16 word 16 word 16 word 16 word unit 16 word 16 word 16 word*
16 word

| *IPv6AddrCompressed6-5 16 word 16 word 16 word 16 word 16 word unit*
| *IPv6AddrCompressed6-6 16 word 16 word 16 word 16 word 16 word unit 16 word*
| *IPv6AddrCompressed6-7 16 word 16 word 16 word 16 word 16 word unit 16 word 16 word*
16 word

| IPv6AddrCompressed7-6 16 word 16 word 16 word 16 word 16 word 16 word 16 word unit

| IPv6AddrCompressed7-7 16 word 16 word 16 word 16 word 16 word 16 word 16 word unit 16 word

| IPv6AddrCompressed8-7 16 word 16 word 16 word 16 word 16 word 16 word 16 word 16 word unit

definition *parse-ipv6-address-compressed* :: ((16 word) option) list \Rightarrow *ipv6addr-syntax-compressed* option where

parse-ipv6-address-compressed as = (case as of
| [None] \Rightarrow Some (IPv6AddrCompressed1-0 ())
| [None, Some a] \Rightarrow Some (IPv6AddrCompressed1-1 () a)
| [None, Some a, Some b] \Rightarrow Some (IPv6AddrCompressed1-2 () a b)
| [None, Some a, Some b, Some c] \Rightarrow Some (IPv6AddrCompressed1-3 () a b c)
| [None, Some a, Some b, Some c, Some d] \Rightarrow Some (IPv6AddrCompressed1-4
() a b c d)
| [None, Some a, Some b, Some c, Some d, Some e] \Rightarrow Some (IPv6AddrCompressed1-5
() a b c d e)
| [None, Some a, Some b, Some c, Some d, Some e, Some f] \Rightarrow Some (IPv6AddrCompressed1-6
() a b c d e f)
| [None, Some a, Some b, Some c, Some d, Some e, Some f, Some g] \Rightarrow Some
(IPv6AddrCompressed1-7 () a b c d e f g)

| [Some a, None] \Rightarrow Some (IPv6AddrCompressed2-1 a ())
| [Some a, None, Some b] \Rightarrow Some (IPv6AddrCompressed2-2 a () b)
| [Some a, None, Some b, Some c] \Rightarrow Some (IPv6AddrCompressed2-3 a () b c)
| [Some a, None, Some b, Some c, Some d] \Rightarrow Some (IPv6AddrCompressed2-4
a () b c d)
| [Some a, None, Some b, Some c, Some d, Some e] \Rightarrow Some (IPv6AddrCompressed2-5
a () b c d e)
| [Some a, None, Some b, Some c, Some d, Some e, Some f] \Rightarrow Some (IPv6AddrCompressed2-6
a () b c d e f)
| [Some a, None, Some b, Some c, Some d, Some e, Some f, Some g] \Rightarrow Some
(IPv6AddrCompressed2-7 a () b c d e f g)

| [Some a, Some b, None] \Rightarrow Some (IPv6AddrCompressed3-2 a b ())
| [Some a, Some b, None, Some c] \Rightarrow Some (IPv6AddrCompressed3-3 a b () c)
| [Some a, Some b, None, Some c, Some d] \Rightarrow Some (IPv6AddrCompressed3-4
a b () c d)
| [Some a, Some b, None, Some c, Some d, Some e] \Rightarrow Some (IPv6AddrCompressed3-5
a b () c d e)
| [Some a, Some b, None, Some c, Some d, Some e, Some f] \Rightarrow Some (IPv6AddrCompressed3-6
a b () c d e f)
| [Some a, Some b, None, Some c, Some d, Some e, Some f, Some g] \Rightarrow Some
(IPv6AddrCompressed3-7 a b () c d e f g)

```

    | [Some a, Some b, Some c, None] ⇒ Some (IPv6AddrCompressed4-3 a b c ())
    | [Some a, Some b, Some c, None, Some d] ⇒ Some (IPv6AddrCompressed4-4
a b c () d)
    | [Some a, Some b, Some c, None, Some d, Some e] ⇒ Some (IPv6AddrCompressed4-5
a b c () d e)
    | [Some a, Some b, Some c, None, Some d, Some e, Some f] ⇒ Some (IPv6AddrCompressed4-6
a b c () d e f)
    | [Some a, Some b, Some c, None, Some d, Some e, Some f, Some g] ⇒ Some
(IPv6AddrCompressed4-7 a b c () d e f g)

    | [Some a, Some b, Some c, Some d, None] ⇒ Some (IPv6AddrCompressed5-4
a b c d ())
    | [Some a, Some b, Some c, Some d, None, Some e] ⇒ Some (IPv6AddrCompressed5-5
a b c d () e)
    | [Some a, Some b, Some c, Some d, None, Some e, Some f] ⇒ Some (IPv6AddrCompressed5-6
a b c d () e f)
    | [Some a, Some b, Some c, Some d, None, Some e, Some f, Some g] ⇒ Some
(IPv6AddrCompressed5-7 a b c d () e f g)

    | [Some a, Some b, Some c, Some d, Some e, None] ⇒ Some (IPv6AddrCompressed6-5
a b c d e ())
    | [Some a, Some b, Some c, Some d, Some e, None, Some f] ⇒ Some (IPv6AddrCompressed6-6
a b c d e () f)
    | [Some a, Some b, Some c, Some d, Some e, None, Some f, Some g] ⇒ Some
(IPv6AddrCompressed6-7 a b c d e () f g)

    | [Some a, Some b, Some c, Some d, Some e, Some f, None] ⇒ Some (IPv6AddrCompressed7-6
a b c d e f ())
    | [Some a, Some b, Some c, Some d, Some e, Some f, None, Some g] ⇒ Some
(IPv6AddrCompressed7-7 a b c d e f () g)

    | [Some a, Some b, Some c, Some d, Some e, Some f, Some g, None] ⇒ Some
(IPv6AddrCompressed8-7 a b c d e f g ())
    | - ⇒ None — invalid ipv6 coppedressed address.
)

```

```

fun ipv6addr-syntax-compressed-to-list :: ipv6addr-syntax-compressed ⇒ ((16 word)
option) list

```

```

where

```

```

    ipv6addr-syntax-compressed-to-list (IPv6AddrCompressed1-0 -) =
        [None]
    | ipv6addr-syntax-compressed-to-list (IPv6AddrCompressed1-1 () a) =
        [None, Some a]
    | ipv6addr-syntax-compressed-to-list (IPv6AddrCompressed1-2 () a b) =
        [None, Some a, Some b]
    | ipv6addr-syntax-compressed-to-list (IPv6AddrCompressed1-3 () a b c) =
        [None, Some a, Some b, Some c]
    | ipv6addr-syntax-compressed-to-list (IPv6AddrCompressed1-4 () a b c d) =

```

$[None, Some\ a, Some\ b, Some\ c, Some\ d]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed1-5\ ()\ a\ b\ c\ d\ e) =$
 $[None, Some\ a, Some\ b, Some\ c, Some\ d, Some\ e]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed1-6\ ()\ a\ b\ c\ d\ e\ f) =$
 $[None, Some\ a, Some\ b, Some\ c, Some\ d, Some\ e,$
Some f]
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed1-7\ ()\ a\ b\ c\ d\ e\ f\ g)$
 $=$
 $[None, Some\ a, Some\ b, Some\ c, Some\ d, Some\ e,$
Some f, Some g]

$| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed2-1\ a\ ()) =$
 $[Some\ a, None]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed2-2\ a\ ()\ b) =$
 $[Some\ a, None, Some\ b]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed2-3\ a\ ()\ b\ c) =$
 $[Some\ a, None, Some\ b, Some\ c]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed2-4\ a\ ()\ b\ c\ d) =$
 $[Some\ a, None, Some\ b, Some\ c, Some\ d]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed2-5\ a\ ()\ b\ c\ d\ e) =$
 $[Some\ a, None, Some\ b, Some\ c, Some\ d, Some\ e]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed2-6\ a\ ()\ b\ c\ d\ e\ f) =$
 $[Some\ a, None, Some\ b, Some\ c, Some\ d, Some\ e,$
Some f]
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed2-7\ a\ ()\ b\ c\ d\ e\ f\ g)$
 $=$
 $[Some\ a, None, Some\ b, Some\ c, Some\ d, Some\ e,$
Some f, Some g]

$| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed3-2\ a\ b\ ()) = [Some$
a, Some b, None]
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed3-3\ a\ b\ ()\ c) =$
 $[Some\ a, Some\ b, None, Some\ c]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed3-4\ a\ b\ ()\ c\ d) =$
 $[Some\ a, Some\ b, None, Some\ c, Some\ d]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed3-5\ a\ b\ ()\ c\ d\ e) =$
 $[Some\ a, Some\ b, None, Some\ c, Some\ d, Some\ e]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed3-6\ a\ b\ ()\ c\ d\ e\ f) =$
 $[Some\ a, Some\ b, None, Some\ c, Some\ d, Some\ e,$
Some f]
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed3-7\ a\ b\ ()\ c\ d\ e\ f\ g)$
 $=$
 $[Some\ a, Some\ b, None, Some\ c, Some\ d, Some\ e,$
Some f, Some g]

$| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed4-3\ a\ b\ c\ ()) =$
 $[Some\ a, Some\ b, Some\ c, None]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed4-4\ a\ b\ c\ ()\ d) =$
 $[Some\ a, Some\ b, Some\ c, None, Some\ d]$

$| \text{ipv6addr-syntax-compressed-to-list } (\text{IPv6AddrCompressed}_{4-5} \ a \ b \ c \ () \ d \ e) =$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ None, \ Some \ d, \ Some \ e]$
 $| \text{ipv6addr-syntax-compressed-to-list } (\text{IPv6AddrCompressed}_{4-6} \ a \ b \ c \ () \ d \ e \ f) =$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ None, \ Some \ d, \ Some \ e,$
 $Some \ f]$
 $| \text{ipv6addr-syntax-compressed-to-list } (\text{IPv6AddrCompressed}_{4-7} \ a \ b \ c \ () \ d \ e \ f \ g)$
 $=$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ None, \ Some \ d, \ Some \ e,$
 $Some \ f, \ Some \ g]$

$| \text{ipv6addr-syntax-compressed-to-list } (\text{IPv6AddrCompressed}_{5-4} \ a \ b \ c \ d \ ()) =$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ Some \ d, \ None]$
 $| \text{ipv6addr-syntax-compressed-to-list } (\text{IPv6AddrCompressed}_{5-5} \ a \ b \ c \ d \ () \ e) =$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ Some \ d, \ None, \ Some \ e]$
 $| \text{ipv6addr-syntax-compressed-to-list } (\text{IPv6AddrCompressed}_{5-6} \ a \ b \ c \ d \ () \ e \ f) =$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ Some \ d, \ None, \ Some \ e,$
 $Some \ f]$
 $| \text{ipv6addr-syntax-compressed-to-list } (\text{IPv6AddrCompressed}_{5-7} \ a \ b \ c \ d \ () \ e \ f \ g)$
 $=$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ Some \ d, \ None, \ Some \ e,$
 $Some \ f, \ Some \ g]$

$| \text{ipv6addr-syntax-compressed-to-list } (\text{IPv6AddrCompressed}_{6-5} \ a \ b \ c \ d \ e \ ()) =$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ Some \ d, \ Some \ e, \ None]$
 $| \text{ipv6addr-syntax-compressed-to-list } (\text{IPv6AddrCompressed}_{6-6} \ a \ b \ c \ d \ e \ () \ f) =$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ Some \ d, \ Some \ e, \ None,$
 $Some \ f]$
 $| \text{ipv6addr-syntax-compressed-to-list } (\text{IPv6AddrCompressed}_{6-7} \ a \ b \ c \ d \ e \ () \ f \ g)$
 $=$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ Some \ d, \ Some \ e, \ None,$
 $Some \ f, \ Some \ g]$

$| \text{ipv6addr-syntax-compressed-to-list } (\text{IPv6AddrCompressed}_{7-6} \ a \ b \ c \ d \ e \ f \ ()) =$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ Some \ d, \ Some \ e, \ Some \ f,$
 $None]$
 $| \text{ipv6addr-syntax-compressed-to-list } (\text{IPv6AddrCompressed}_{7-7} \ a \ b \ c \ d \ e \ f \ () \ g)$
 $=$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ Some \ d, \ Some \ e, \ Some \ f,$
 $None, \ Some \ g]$

$| \text{ipv6addr-syntax-compressed-to-list } (\text{IPv6AddrCompressed}_{8-7} \ a \ b \ c \ d \ e \ f \ g \ ())$
 $=$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ Some \ d, \ Some \ e, \ Some \ f,$
 $Some \ g, \ None]$

lemma *parse-ipv6-address-compressed-exists:*

obtains *ss* where *parse-ipv6-address-compressed ss = Some ipv6-syntax*

proof

define *ss* **where** *ss* = *ipv6addr-syntax-compressed-to-list ipv6-syntax*
thus *parse-ipv6-address-compressed ss* = *Some ipv6-syntax*
by (*cases ipv6-syntax; simp add: parse-ipv6-address-compressed-def*)
qed

lemma *parse-ipv6-address-compressed-identity*:

parse-ipv6-address-compressed (ipv6addr-syntax-compressed-to-list (ipv6-syntax))
= *Some ipv6-syntax*
by(*cases ipv6-syntax; simp add: parse-ipv6-address-compressed-def*)

lemma *parse-ipv6-address-compressed-someE*:

assumes *parse-ipv6-address-compressed as* = *Some ipv6*

obtains

as = [*None*] *ipv6* = (*IPv6AddrCompressed1-0* ()) |
a **where** *as* = [*None, Some a*] *ipv6* = (*IPv6AddrCompressed1-1* () *a*) |
a b **where** *as* = [*None, Some a, Some b*] *ipv6* = (*IPv6AddrCompressed1-2* ()
a b) |
a b c **where** *as* = [*None, Some a, Some b, Some c*] *ipv6* = (*IPv6AddrCompressed1-3*
(*a b c*) |
a b c d **where** *as* = [*None, Some a, Some b, Some c, Some d*] *ipv6* =
(*IPv6AddrCompressed1-4* () *a b c d*) |
a b c d e **where** *as* = [*None, Some a, Some b, Some c, Some d, Some e*] *ipv6*
= (*IPv6AddrCompressed1-5* () *a b c d e*) |
a b c d e f **where** *as* = [*None, Some a, Some b, Some c, Some d, Some e,*
Some f] *ipv6* = (*IPv6AddrCompressed1-6* () *a b c d e f*) |
a b c d e f g **where** *as* = [*None, Some a, Some b, Some c, Some d, Some e,*
Some f, Some g] *ipv6* = (*IPv6AddrCompressed1-7* () *a b c d e f g*) |

a **where** *as* = [*Some a, None*] *ipv6* = (*IPv6AddrCompressed2-1* *a* ()) |
a b **where** *as* = [*Some a, None, Some b*] *ipv6* = (*IPv6AddrCompressed2-2* *a* ()
b) |
a b c **where** *as* = [*Some a, None, Some b, Some c*] *ipv6* = (*IPv6AddrCompressed2-3*
a () *b c*) |
a b c d **where** *as* = [*Some a, None, Some b, Some c, Some d*] *ipv6* =
(*IPv6AddrCompressed2-4* *a* () *b c d*) |
a b c d e **where** *as* = [*Some a, None, Some b, Some c, Some d, Some e*] *ipv6*
= (*IPv6AddrCompressed2-5* *a* () *b c d e*) |
a b c d e f **where** *as* = [*Some a, None, Some b, Some c, Some d, Some e,*
Some f] *ipv6* = (*IPv6AddrCompressed2-6* *a* () *b c d e f*) |
a b c d e f g **where** *as* = [*Some a, None, Some b, Some c, Some d, Some e,*
Some f, Some g] *ipv6* = (*IPv6AddrCompressed2-7* *a* () *b c d e f g*) |

a b **where** *as* = [*Some a, Some b, None*] *ipv6* = (*IPv6AddrCompressed3-2* *a b*
(*a b*)) |
a b c **where** *as* = [*Some a, Some b, None, Some c*] *ipv6* = (*IPv6AddrCompressed3-3*
a b () *c*) |
a b c d **where** *as* = [*Some a, Some b, None, Some c, Some d*] *ipv6* =

$(IPv6AddrCompressed3-4\ a\ b\ ()\ c\ d) \mid$
 $a\ b\ c\ d\ e\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ None,\ Some\ c,\ Some\ d,\ Some\ e]\ ipv6$
 $= (IPv6AddrCompressed3-5\ a\ b\ ()\ c\ d\ e) \mid$
 $a\ b\ c\ d\ e\ f\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ None,\ Some\ c,\ Some\ d,\ Some\ e,$
 $Some\ f]\ ipv6 = (IPv6AddrCompressed3-6\ a\ b\ ()\ c\ d\ e\ f) \mid$
 $a\ b\ c\ d\ e\ f\ g\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ None,\ Some\ c,\ Some\ d,\ Some\ e,$
 $Some\ f,\ Some\ g]\ ipv6 = (IPv6AddrCompressed3-7\ a\ b\ ()\ c\ d\ e\ f\ g) \mid$

$a\ b\ c\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ None]\ ipv6 = (IPv6AddrCompressed4-3$
 $a\ b\ c\ ()) \mid$
 $a\ b\ c\ d\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ None,\ Some\ d]\ ipv6 =$
 $(IPv6AddrCompressed4-4\ a\ b\ c\ ()\ d) \mid$
 $a\ b\ c\ d\ e\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ None,\ Some\ d,\ Some\ e]\ ipv6$
 $= (IPv6AddrCompressed4-5\ a\ b\ c\ ()\ d\ e) \mid$
 $a\ b\ c\ d\ e\ f\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ None,\ Some\ d,\ Some\ e,$
 $Some\ f]\ ipv6 = (IPv6AddrCompressed4-6\ a\ b\ c\ ()\ d\ e\ f) \mid$
 $a\ b\ c\ d\ e\ f\ g\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ None,\ Some\ d,\ Some\ e,$
 $Some\ f,\ Some\ g]\ ipv6 = (IPv6AddrCompressed4-7\ a\ b\ c\ ()\ d\ e\ f\ g) \mid$

$a\ b\ c\ d\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ None]\ ipv6 =$
 $(IPv6AddrCompressed5-4\ a\ b\ c\ d\ ()) \mid$
 $a\ b\ c\ d\ e\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ None,\ Some\ e]\ ipv6$
 $= (IPv6AddrCompressed5-5\ a\ b\ c\ d\ ()\ e) \mid$
 $a\ b\ c\ d\ e\ f\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ None,\ Some\ e,$
 $Some\ f]\ ipv6 = (IPv6AddrCompressed5-6\ a\ b\ c\ d\ ()\ e\ f) \mid$
 $a\ b\ c\ d\ e\ f\ g\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ None,\ Some\ e,$
 $Some\ f,\ Some\ g]\ ipv6 = (IPv6AddrCompressed5-7\ a\ b\ c\ d\ ()\ e\ f\ g) \mid$

$a\ b\ c\ d\ e\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ Some\ e,\ None]\ ipv6$
 $= (IPv6AddrCompressed6-5\ a\ b\ c\ d\ e\ ()) \mid$
 $a\ b\ c\ d\ e\ f\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ Some\ e,\ None,$
 $Some\ f]\ ipv6 = (IPv6AddrCompressed6-6\ a\ b\ c\ d\ e\ ()\ f) \mid$
 $a\ b\ c\ d\ e\ f\ g\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ Some\ e,\ None,$
 $Some\ f,\ Some\ g]\ ipv6 = (IPv6AddrCompressed6-7\ a\ b\ c\ d\ e\ ()\ f\ g) \mid$

$a\ b\ c\ d\ e\ f\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ Some\ e,\ Some\ f,$
 $None]\ ipv6 = (IPv6AddrCompressed7-6\ a\ b\ c\ d\ e\ f\ ()) \mid$
 $a\ b\ c\ d\ e\ f\ g\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ Some\ e,\ Some\ f,$
 $None,\ Some\ g]\ ipv6 = (IPv6AddrCompressed7-7\ a\ b\ c\ d\ e\ f\ ()\ g) \mid$

$a\ b\ c\ d\ e\ f\ g\ \mathbf{where}\ as = [Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ Some\ e,\ Some\ f,$
 $Some\ g,\ None]\ ipv6 = (IPv6AddrCompressed8-7\ a\ b\ c\ d\ e\ f\ g\ ())$
using *assms*
unfolding *parse-ipv6-address-compressed-def*
by (*auto split: list.split-asm option.split-asm*)

lemma *parse-ipv6-address-compressed-identity2*:
 $ipv6addr-syntax-compressed-to-list\ ipv6-syntax = ls \longleftrightarrow$
 $(parse-ipv6-address-compressed\ ls) = Some\ ipv6-syntax$

```

      (is ?lhs = ?rhs)
proof
  assume ?rhs
  thus ?lhs
    by (auto elim: parse-ipv6-address-compressed-someE)
next
  assume ?lhs
  thus ?rhs
    by (cases ipv6-syntax) (auto simp: parse-ipv6-address-compressed-def)
qed

```

Valid IPv6 compressed notation:

- at most one omission
- at most 7 pieces

lemma *RFC-4291-format: parse-ipv6-address-compressed as ≠ None* \longleftrightarrow
 $\text{length (filter } (\lambda p. p = \text{None}) \text{ as)} = 1 \wedge \text{length (filter } (\lambda p. p \neq \text{None}) \text{ as)} \leq 7$

```

      (is ?lhs = ?rhs)
proof
  assume ?lhs
  then obtain addr where parse-ipv6-address-compressed as = Some addr
    by blast
  thus ?rhs
    by (elim parse-ipv6-address-compressed-someE; simp)
next
  assume ?rhs
  thus ?lhs
    unfolding parse-ipv6-address-compressed-def
    by (auto split: option.split list.split if-split-asm)
qed

```

3. An alternative form that is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 nodes is `x:x:x:x:x:d.d.d.d`, where the 'x's are the hexadecimal values of the six high-order 16-bit pieces of the address, and the 'd's are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation). Examples:

```

0:0:0:0:0:0:13.1.68.3
0:0:0:0:0:FFFF:129.144.52.38

```

or in compressed form:

```

::13.1.68.3
::FFFF:129.144.52.38

```

This is currently not supported by our library!

5.2 Semantics

```

context
  includes bit-operations-syntax
begin

  fun ipv6preferred-to-int :: ipv6addr-syntax  $\Rightarrow$  ipv6addr where
    ipv6preferred-to-int (IPv6AddrPreferred a b c d e f g h) = (ucast a << (16 *
7)) OR
                                     (ucast b << (16 * 6)) OR
                                     (ucast c << (16 * 5)) OR
                                     (ucast d << (16 * 4)) OR
                                     (ucast e << (16 * 3)) OR
                                     (ucast f << (16 * 2)) OR
                                     (ucast g << (16 * 1)) OR
                                     (ucast h << (16 * 0))

  lemma ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0xDB8 0x0 0x0 0x8 0x800
0x200C 0x417A) =
    42540766411282592856906245548098208122 by eval

  lemma ipv6preferred-to-int (IPv6AddrPreferred 0xFF01 0x0 0x0 0x0 0x0 0x0 0x0
0x101) =
    338958331222012082418099330867817087233 by eval

  declare ipv6preferred-to-int.simps[simp del]

  definition int-to-ipv6preferred :: ipv6addr  $\Rightarrow$  ipv6addr-syntax where
    int-to-ipv6preferred i = IPv6AddrPreferred (ucast ((i AND 0xFFFF0000000000000000000000000000)
>> 16*7))
                                     (ucast ((i AND 0xFFFF0000000000000000000000000000)
>> 16*6))
                                     (ucast ((i AND 0xFFFF0000000000000000000000000000)
>> 16*5))
                                     (ucast ((i AND 0xFFFF0000000000000000000000000000)
>> 16*4))
                                     (ucast ((i AND 0xFFFF000000000000) >>
16*3))
                                     (ucast ((i AND 0xFFFF00000000) >> 16*2))
                                     (ucast ((i AND 0xFFFF0000) >> 16*1))
                                     (ucast ((i AND 0xFFFF)))

  lemma int-to-ipv6preferred 42540766411282592856906245548098208122 =
    IPv6AddrPreferred 0x2001 0xDB8 0x0 0x0 0x8 0x800 0x200C 0x417A by
eval

  lemma word128-masks-ipv6pieces:
    (0xFFFF0000000000000000000000000000::ipv6addr) = (mask 16) << 112
    (0xFFFF0000000000000000000000000000::ipv6addr) = (mask 16) << 96
    (0xFFFF0000000000000000000000000000::ipv6addr) = (mask 16) << 80

```

```

(0xFFFF0000000000000000000000000000::ipv6addr) = (mask 16) << 64
(0xFFFF0000000000000000000000000000::ipv6addr) = (mask 16) << 48
(0xFFFF0000000000000000000000000000::ipv6addr) = (mask 16) << 32
(0xFFFF00000::ipv6addr) = (mask 16) << 16
(0xFFFF::ipv6addr) = (mask 16)
by (simp-all add: mask-eq)

```

Correctness: round trip property one

lemma *ipv6preferred-to-int-int-to-ipv6preferred*:

ipv6preferred-to-int (int-to-ipv6preferred ip) = ip

proof –

have *and-mask-shift-helper*: $w \text{ AND } (\text{mask } m \ll n) \gg n \ll n = w \text{ AND } (\text{mask } m \ll n)$

for $m n::\text{nat}$ **and** $w::\text{ipv6addr}$

by (*metis is-aligned-shift is-aligned-shiftr-shiftr-shiftr-and-eq-shiftr*)

have *ucast-ipv6-piece-rule*:

$\text{length } (\text{dropWhile Not } (\text{to-bl } w)) \leq 16 \implies (\text{ucast}::16 \text{ word} \Rightarrow 128 \text{ word})$

$((\text{ucast}::128 \text{ word} \Rightarrow 16 \text{ word}) w) = w$

for $w::\text{ipv6addr}$

by(*rule ucast-short-ucast-long-ingoreLeadingZero*) (*simp-all*)

have *ucast-ipv6-piece*: $16 \leq 128 - n \implies$

$(\text{ucast}::16 \text{ word} \Rightarrow 128 \text{ word}) ((\text{ucast}::128 \text{ word} \Rightarrow 16 \text{ word}) (w \text{ AND } (\text{mask } 16 \ll n) \gg n)) \ll n = w \text{ AND } (\text{mask } 16 \ll n)$

for $w::\text{ipv6addr}$ **and** $n::\text{nat}$

apply(*subst ucast-ipv6-piece-rule*)

apply(*rule length-drop-mask-inner*)

apply(*simp; fail*)

apply(*subst and-mask-shift-helper*)

apply *simp*

done

have *ucast16-ucast128-masks-highest-bits*:

$(\text{ucast } ((\text{ucast}::\text{ipv6addr} \Rightarrow 16 \text{ word}) (ip \text{ AND } 0xFFFF0000000000000000000000000000 \gg 112)) \ll 112) =$

$(ip \text{ AND } 0xFFFF0000000000000000000000000000)$

$(\text{ucast } ((\text{ucast}::\text{ipv6addr} \Rightarrow 16 \text{ word}) (ip \text{ AND } 0xFFFF0000000000000000000000000000 \gg 96)) \ll 96) =$

$ip \text{ AND } 0xFFFF0000000000000000000000000000$

$(\text{ucast } ((\text{ucast}::\text{ipv6addr} \Rightarrow 16 \text{ word}) (ip \text{ AND } 0xFFFF0000000000000000000000000000 \gg 80)) \ll 80) =$

$ip \text{ AND } 0xFFFF0000000000000000000000000000$

$(\text{ucast } ((\text{ucast}::\text{ipv6addr} \Rightarrow 16 \text{ word}) (ip \text{ AND } 0xFFFF00000000000000000000 \gg 64)) \ll 64) =$

$ip \text{ AND } 0xFFFF0000000000000000000000000000$

$(\text{ucast } ((\text{ucast}::\text{ipv6addr} \Rightarrow 16 \text{ word}) (ip \text{ AND } 0xFFFF000000000000 \gg 48)) \ll 48) =$

$ip \text{ AND } 0xFFFF0000000000000000000000000000$

$(\text{ucast } ((\text{ucast}::\text{ipv6addr} \Rightarrow 16 \text{ word}) (ip \text{ AND } 0xFFFF00000000 \gg 32)) \ll 32) =$

```

      ip AND 0xFFFF00000000
      (ucast ((ucast::ipv6addr ⇒ 16 word) (ip AND 0xFFFF0000 >> 16)) << 16)
=
      ip AND 0xFFFF0000
      apply (simp-all only: word128-masks-ipv6pieces ucast-ipv6-piece and-mask2
word-size bit-eq-iff bit-simps comp-def)
      apply auto
      done

```

```

have ucast16-ucast128-masks-highest-bits0:
  (ucast ((ucast::ipv6addr ⇒ 16 word) (ip AND 0xFFFF))) = ip AND 0xFFFF
  apply (simp only: word128-masks-ipv6pieces flip: take-bit-eq-mask)
  apply (simp add: unsigned-ucast-eq)
  done

```

```

have mask-len-word:n = (LENGTH('a)) ⇒ w AND mask n = w
  for n and w::'a::len word by (simp add: mask-eq-iff)

```

```

have ipv6addr-16word-pieces-compose-or:

```

```

  ip && (mask 16 << 112) ||
  ip && (mask 16 << 96) ||
  ip && (mask 16 << 80) ||
  ip && (mask 16 << 64) ||
  ip && (mask 16 << 48) ||
  ip && (mask 16 << 32) ||
  ip && (mask 16 << 16) ||
  ip && mask 16 =
  ip

```

```

  apply(subst word-ao-dist2[symmetric])+
  apply(simp add: mask-numeral)
  apply(subst mask128)
  apply(rule mask-len-word)
  apply simp
  done

```

```

show ?thesis

```

```

  apply (simp add: ipv6preferred-to-int.simps int-to-ipv6preferred-def shiftr-def
shiftr-def)
  apply (simp only: word128-masks-ipv6pieces flip: take-bit-eq-mask)
  apply (simp add: unsigned-ucast-eq push-bit-take-bit)
  using ipv6addr-16word-pieces-compose-or
  apply (simp add: take-bit-push-bit slice-eq-mask)
  apply (simp add: take-bit-eq-mask shiftr-def push-bit-mask-eq)
  done

```

```

qed

```

Correctness: round trip property two

```

lemma int-to-ipv6preferred-ipv6preferred-to-int: int-to-ipv6preferred (ipv6preferred-to-int
ip) = ip

```

```

proof –
  note ucast-shift-simps=helper-masked-ucast-generic helper-masked-ucast-reverse-generic
        helper-masked-ucast-generic[where n=0, simplified]
        helper-masked-ucast-equal-generic
  note ucast-simps=helper-masked-ucast-reverse-generic[where m=0, simplified]
        helper-masked-ucast-equal-generic[where n=0, simplified]
  show ?thesis
  apply (cases ip, rename-tac a b c d e f g h)
  apply (simp add: ipv6preferred-to-int.simps int-to-ipv6preferred-def)
  apply (simp add: word128-masks-ipv6pieces)
  apply (simp add: word-ao-dist ucast-shift-simps ucast-simps)
  done
qed

```

compressed to preferred format

```

fun ipv6addr-c2p :: ipv6addr-syntax-compressed ⇒ ipv6addr-syntax where
  ipv6addr-c2p (IPv6AddrCompressed1-0 ()) = IPv6AddrPreferred 0 0 0 0 0 0 0
  0
  | ipv6addr-c2p (IPv6AddrCompressed1-1 () h) = IPv6AddrPreferred 0 0 0 0 0 0
  0 h
  | ipv6addr-c2p (IPv6AddrCompressed1-2 () g h) = IPv6AddrPreferred 0 0 0 0 0
  0 g h
  | ipv6addr-c2p (IPv6AddrCompressed1-3 () f g h) = IPv6AddrPreferred 0 0 0 0
  0 f g h
  | ipv6addr-c2p (IPv6AddrCompressed1-4 () e f g h) = IPv6AddrPreferred 0 0 0
  0 e f g h
  | ipv6addr-c2p (IPv6AddrCompressed1-5 () d e f g h) = IPv6AddrPreferred 0 0
  0 d e f g h
  | ipv6addr-c2p (IPv6AddrCompressed1-6 () c d e f g h) = IPv6AddrPreferred 0 0
  c d e f g h
  | ipv6addr-c2p (IPv6AddrCompressed1-7 () b c d e f g h) = IPv6AddrPreferred 0
  b c d e f g h

  | ipv6addr-c2p (IPv6AddrCompressed2-1 a ()) = IPv6AddrPreferred a 0 0 0 0 0
  0 0
  | ipv6addr-c2p (IPv6AddrCompressed2-2 a () h) = IPv6AddrPreferred a 0 0 0 0
  0 0 h
  | ipv6addr-c2p (IPv6AddrCompressed2-3 a () g h) = IPv6AddrPreferred a 0 0 0
  0 0 g h
  | ipv6addr-c2p (IPv6AddrCompressed2-4 a () f g h) = IPv6AddrPreferred a 0 0 0
  0 f g h
  | ipv6addr-c2p (IPv6AddrCompressed2-5 a () e f g h) = IPv6AddrPreferred a 0 0
  0 e f g h
  | ipv6addr-c2p (IPv6AddrCompressed2-6 a () d e f g h) = IPv6AddrPreferred a 0
  0 d e f g h
  | ipv6addr-c2p (IPv6AddrCompressed2-7 a () c d e f g h) = IPv6AddrPreferred a
  0 c d e f g h

  | ipv6addr-c2p (IPv6AddrCompressed3-2 a b ()) = IPv6AddrPreferred a b 0 0 0

```

$0\ 0\ 0$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed3-3 } a\ b\ ()\ h) = \text{IPv6AddrPreferred } a\ b\ 0\ 0$
 $0\ 0\ 0\ h$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed3-4 } a\ b\ ()\ g\ h) = \text{IPv6AddrPreferred } a\ b\ 0$
 $0\ 0\ 0\ g\ h$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed3-5 } a\ b\ ()\ f\ g\ h) = \text{IPv6AddrPreferred } a\ b\ 0$
 $0\ 0\ f\ g\ h$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed3-6 } a\ b\ ()\ e\ f\ g\ h) = \text{IPv6AddrPreferred } a\ b$
 $0\ 0\ e\ f\ g\ h$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed3-7 } a\ b\ ()\ d\ e\ f\ g\ h) = \text{IPv6AddrPreferred } a$
 $b\ 0\ d\ e\ f\ g\ h$

$| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed4-3 } a\ b\ c\ ()) = \text{IPv6AddrPreferred } a\ b\ c\ 0\ 0$
 $0\ 0\ 0$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed4-4 } a\ b\ c\ ()\ h) = \text{IPv6AddrPreferred } a\ b\ c$
 $0\ 0\ 0\ 0\ h$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed4-5 } a\ b\ c\ ()\ g\ h) = \text{IPv6AddrPreferred } a\ b\ c$
 $0\ 0\ 0\ g\ h$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed4-6 } a\ b\ c\ ()\ f\ g\ h) = \text{IPv6AddrPreferred } a\ b$
 $c\ 0\ 0\ f\ g\ h$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed4-7 } a\ b\ c\ ()\ e\ f\ g\ h) = \text{IPv6AddrPreferred } a$
 $b\ c\ 0\ e\ f\ g\ h$

$| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed5-4 } a\ b\ c\ d\ ()) = \text{IPv6AddrPreferred } a\ b\ c$
 $d\ 0\ 0\ 0\ 0$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed5-5 } a\ b\ c\ d\ ()\ h) = \text{IPv6AddrPreferred } a\ b$
 $c\ d\ 0\ 0\ 0\ h$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed5-6 } a\ b\ c\ d\ ()\ g\ h) = \text{IPv6AddrPreferred } a$
 $b\ c\ d\ 0\ 0\ g\ h$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed5-7 } a\ b\ c\ d\ ()\ f\ g\ h) = \text{IPv6AddrPreferred } a$
 $b\ c\ d\ 0\ f\ g\ h$

$| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed6-5 } a\ b\ c\ d\ e\ ()) = \text{IPv6AddrPreferred } a\ b\ c$
 $d\ e\ 0\ 0\ 0$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed6-6 } a\ b\ c\ d\ e\ ()\ h) = \text{IPv6AddrPreferred } a$
 $b\ c\ d\ e\ 0\ 0\ h$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed6-7 } a\ b\ c\ d\ e\ ()\ g\ h) = \text{IPv6AddrPreferred}$
 $a\ b\ c\ d\ e\ 0\ g\ h$

$| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed7-6 } a\ b\ c\ d\ e\ f\ ()) = \text{IPv6AddrPreferred } a\ b$
 $c\ d\ e\ f\ 0\ 0$
 $| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed7-7 } a\ b\ c\ d\ e\ f\ ()\ h) = \text{IPv6AddrPreferred } a$
 $b\ c\ d\ e\ f\ 0\ h$

$| \text{ipv6addr-c2p} (\text{IPv6AddrCompressed8-7 } a\ b\ c\ d\ e\ f\ g\ ()) = \text{IPv6AddrPreferred } a$
 $b\ c\ d\ e\ f\ g\ 0$

definition *ipv6-unparsed-compressed-to-preferred* :: ((16 word) option) list \Rightarrow *ipv6addr-syntax*

option where

```
ipv6-unparsed-compressed-to-preferred ls = (  
  if  
    length (filter (λp. p = None) ls) ≠ 1 ∨ length (filter (λp. p ≠ None) ls) > 7  
  then  
    None  
  else  
    let  
      before-omission = map the (takeWhile (λx. x ≠ None) ls);  
      after-omission = map the (drop 1 (dropWhile (λx. x ≠ None) ls));  
      num-omissions = 8 - (length before-omission + length after-omission);  
      expanded = before-omission @ (replicate num-omissions 0) @ after-omission  
    in  
      case expanded of [a,b,c,d,e,f,g,h] ⇒ Some (IPv6AddrPreferred a b c d e f g  
h)  
      | - ⇒ None  
    )
```

lemma *ipv6-unparsed-compressed-to-preferred*
[Some 0x2001, Some 0xDB8, None, Some 0x8, Some 0x800, Some 0x200C,
Some 0x417A]
= Some (IPv6AddrPreferred 0x2001 0xDB8 0 0 8 0x800 0x200C 0x417A) **by**
eval

lemma *ipv6-unparsed-compressed-to-preferred* [None] = Some (IPv6AddrPreferred
0 0 0 0 0 0 0 0) **by** *eval*

lemma *ipv6-unparsed-compressed-to-preferred* [] = None **by** *eval*

lemma *ipv6-unparsed-compressed-to-preferred-identity1*:
ipv6-unparsed-compressed-to-preferred (*ipv6addr-syntax-compressed-to-list* *ipv6com-*
pressed) = Some *ipv6preferred*
↔ *ipv6addr-c2p* *ipv6compressed* = *ipv6preferred*
by (cases *ipv6compressed*) (*simp-all* add: *ipv6-unparsed-compressed-to-preferred-def*
numeral-eq-Suc)

lemma *ipv6-unparsed-compressed-to-preferred-identity2*:
ipv6-unparsed-compressed-to-preferred ls = Some *ipv6preferred*
↔ (∃ *ipv6compressed*. *parse-ipv6-address-compressed* ls = Some *ipv6com-*
pressed ∧
ipv6addr-c2p *ipv6compressed* = *ipv6preferred*)

apply(*rule iffI*)
apply(*subgoal-tac* *parse-ipv6-address-compressed* ls ≠ None)
prefer 2
apply(*subst* *RFC-4291-format*)
apply(*simp* add: *ipv6-unparsed-compressed-to-preferred-def* *split: if-split-asm*;
fail)
apply(*simp*)

```

apply(erule exE, rename-tac ipv6compressed)
apply(rule-tac x=ipv6compressed in exI)
apply(simp)
apply(subgoal-tac (ipv6addr-syntax-compressed-to-list ipv6compressed = ls))
prefer 2
using parse-ipv6-address-compressed-identity2 apply presburger
using ipv6-unparsed-compressed-to-preferred-identity1 apply blast
apply(erule exE, rename-tac ipv6compressed)
apply(subgoal-tac (ipv6addr-syntax-compressed-to-list ipv6compressed = ls))
prefer 2
using parse-ipv6-address-compressed-identity2 apply presburger
using ipv6-unparsed-compressed-to-preferred-identity1 apply blast
done

end

```

5.3 IPv6 Pretty Printing (converting to compressed format)

RFC5952:

4. A Recommendation for IPv6 Text Representation

A recommendation for a canonical text representation format of IPv6 addresses is presented in this section. The recommendation in this document is one that complies fully with [RFC4291], is implemented by various operating systems, and is human friendly. The recommendation in this section SHOULD be followed by systems when generating an address to be represented as text, but all implementations MUST accept and be able to handle any legitimate [RFC4291] format. It is advised that humans also follow these recommendations when spelling an address.

4.1. Handling Leading Zeros in a 16-Bit Field

Leading zeros MUST be suppressed. For example, 2001:0db8::0001 is not acceptable and must be represented as 2001:db8::1. A single 16-bit 0000 field MUST be represented as 0.

4.2. "::" Usage

4.2.1. Shorten as Much as Possible

The use of the symbol "::" MUST be used to its maximum capability. For example, 2001:db8:0:0:0:0:2:1 must be shortened to 2001:db8::2:1. Likewise, 2001:db8::0:1 is not acceptable, because the symbol "::" could have been used to produce a shorter representation 2001:db8::1.

4.2.2. Handling One 16-Bit 0 Field

The symbol "::" MUST NOT be used to shorten just one 16-bit 0 field.

For example, the representation 2001:db8:0:1:1:1:1:1 is correct, but 2001:db8::1:1:1:1:1 is not correct.

4.2.3. Choice in Placement of "::"

When there is an alternative choice in the placement of a "::", the longest run of consecutive 16-bit 0 fields MUST be shortened (i.e., the sequence with three consecutive zero fields is shortened in 2001:0:0:1:0:0:0:1). When the length of the consecutive 16-bit 0 fields are equal (i.e., 2001:db8:0:0:1:0:0:1), the first sequence of zero bits MUST be shortened. For example, 2001:db8::1:0:0:1 is correct representation.

4.3. Lowercase

The characters "a", "b", "c", "d", "e", and "f" in an IPv6 address MUST be represented in lowercase.

See `IP_Address_toString.thy` for examples and test cases.

context

begin

private function *goup-by-zeros* :: 16 word list \Rightarrow 16 word list list **where**

goup-by-zeros [] = [] |

goup-by-zeros (x#xs) = (

if x = 0

then takeWhile ($\lambda x. x = 0$) (x#xs) # (*goup-by-zeros* (dropWhile ($\lambda x. x = 0$) xs))

else [x]#(*goup-by-zeros* xs))

by(*pat-completeness*, *auto*)

termination *goup-by-zeros*

apply(*relation measure* ($\lambda xs. \text{length } xs$))

apply(*simp-all*)

by (*simp add: le-imp-less-Suc length-dropWhile-le*)

private lemma *goup-by-zeros* [0,1,2,3,0,0,0,0,3,4,0,0,0,2,0,0,2,0,3,0] =

[[0], [1], [2], [3], [0, 0, 0, 0], [3], [4], [0, 0, 0], [2], [0, 0], [2], [0], [3], [0]]

by *eval*

private lemma *concat* (*goup-by-zeros* ls) = ls

by(*induction ls rule:goup-by-zeros.induct*) *simp+*

private lemma [] \notin set (*goup-by-zeros* ls)

by(*induction ls rule:goup-by-zeros.induct*) *simp+*

private primrec *List-replace1* :: 'a \Rightarrow 'a \Rightarrow 'a list \Rightarrow 'a list **where**

List-replace1 - - [] = [] |

List-replace1 a b (x#xs) = (if a = x then b#xs else x#*List-replace1* a b xs)

```

private lemma List-replace1 a a ls = ls
  by(induction ls) simp-all

private lemma a ∉ set ls ⇒ List-replace1 a b ls = ls
  by(induction ls) simp-all

private lemma a ∈ set ls ⇒ b ∈ set (List-replace1 a b ls)
  apply(induction ls)
  apply(simp)
  apply(simp)
  by blast

private fun List-explode :: 'a list list ⇒ ('a option) list where
  List-explode [] = [] |
  List-explode ([]#xs) = None#List-explode xs |
  List-explode (xs1#xs2) = map Some xs1@List-explode xs2

private lemma List-explode [[0::int], [2,3], [], [3,4]] = [Some 0, Some 2, Some 3, None, Some 3, Some 4]
  by eval

private lemma List-explode-def:
  List-explode xss = concat (map (λxs. if xs = [] then [None] else map Some xs) xss)
  by(induction xss rule: List-explode.induct) simp+

private lemma List-explode-no-empty: [] ∉ set xss ⇒ List-explode xss = map Some (concat xss)
  by(induction xss rule: List-explode.induct) simp+

private lemma List-explode-replace1: [] ∉ set xss ⇒ foo ∈ set xss ⇒
  List-explode (List-replace1 foo [] xss) =
  map Some (concat (takeWhile (λxs. xs ≠ foo) xss)) @ [None] @
  map Some (concat (tl (dropWhile (λxs. xs ≠ foo) xss)))
  apply(induction xss rule: List-explode.induct)
  apply(simp; fail)
  apply(simp; fail)
  apply(simp)
  apply safe
  apply(simp-all add: List-explode-no-empty)
  done

fun ipv6-preferred-to-compressed :: ipv6addr-syntax ⇒ ((16 word) option) list
where
  ipv6-preferred-to-compressed (IPv6AddrPreferred a b c d e f g h) = (
  let lss = group-by-zeros [a,b,c,d,e,f,g,h];
  max-zero-seq = foldr (λxs. max (length xs)) lss 0;
  shortened = if max-zero-seq > 1 then List-replace1 (replicate max-zero-seq
  0) [] lss else lss

```

```

    in
      List-explode shortened
    )
  declare ipv6-preferred-to-compressed.simps[simp del]

  private lemma foldr-max-length: foldr (λxs. max (length xs)) lss n = fold max
  (map length lss) n
  apply(subst List.foldr-fold)
  apply fastforce
  apply(induction lss arbitrary: n)
  apply(simp; fail)
  apply(simp)
  done

  private lemma List-explode-goup-by-zeros: List-explode (goup-by-zeros xs) =
  map Some xs
  apply(induction xs rule: goup-by-zeros.induct)
  apply(simp; fail)
  apply(simp)
  apply(safe)
  apply(simp)
  by (metis map-append takeWhile-dropWhile-id)

  private definition max-zero-streak xs ≡ foldr (λxs. max (length xs)) (goup-by-zeros
  xs) 0

  private lemma max-zero-streak-def2: max-zero-streak xs = fold max (map length
  (goup-by-zeros xs)) 0
  unfolding max-zero-streak-def
  by(simp add: foldr-max-length)

  private lemma ipv6-preferred-to-compressed-pull-out-if:
  ipv6-preferred-to-compressed (IPv6AddrPreferred a b c d e f g h) = (
  if max-zero-streak [a,b,c,d,e,f,g,h] > 1 then
    List-explode (List-replace1 (replicate (max-zero-streak [a,b,c,d,e,f,g,h]) 0) [])
  (goup-by-zeros [a,b,c,d,e,f,g,h]))
  else
    map Some [a,b,c,d,e,f,g,h]
  )
  apply (simp only: ipv6-preferred-to-compressed.simps Let-def max-zero-streak-def
  List-explode-goup-by-zeros)
  using List-explode-goup-by-zeros by presburger

  private lemma ipv6-preferred-to-compressed (IPv6AddrPreferred 0 0 0 0 0 0 0
  0) = [None] by eval
  private lemma ipv6-preferred-to-compressed (IPv6AddrPreferred 0x2001 0xDB8
  0 0 8 0x800 0x200C 0x417A) =

```

[Some 0x2001, Some 0xDB8, None, Some 8, Some 0x800,
Some 0x200C, Some 0x417A] **by eval**

private lemma *ipv6-preferred-to-compressed* (IPv6AddrPreferred 0x2001 0xDB8
0 3 8 0x800 0x200C 0x417A) =
[Some 0x2001, Some 0xDB8, Some 0, Some 3, Some 8, Some 0x800,
Some 0x200C, Some 0x417A] **by eval**

lemma *ipv6-preferred-to-compressed-RFC-4291-format*:
ipv6-preferred-to-compressed ip = as \implies
 $\text{length (filter } (\lambda p. p = \text{None}) \text{ as)} = 0 \wedge \text{length as} = 8$
 \vee
 $\text{length (filter } (\lambda p. p = \text{None}) \text{ as)} = 1 \wedge \text{length (filter } (\lambda p. p \neq \text{None}) \text{ as)}$
 ≤ 7
apply(cases ip)
apply(simp add: *ipv6-preferred-to-compressed-pull-out-if*)
apply(simp only: *split: if-split-asm*)
subgoal for a b c d e f g h
apply(rule *disjI2*)
apply(case-tac a=0, case-tac [!] b=0, case-tac [!] c=0, case-tac [!] d=0,
case-tac [!] e=0, case-tac [!] f=0, case-tac [!] g=0, case-tac [!] h=0)
by(auto simp add: *max-zero-streak-def*)
subgoal
apply(rule *disjI1*)
apply(simp)
by force
done

— Idea for the following proof:

private lemma *ipv6-preferred-to-compressed* (IPv6AddrPreferred a b c d e f g
h) = None#xs \implies
xs = map Some (dropWhile ($\lambda x. x=0$) [a,b,c,d,e,f,g,h])
apply(case-tac a=0, case-tac [!] b=0, case-tac [!] c=0, case-tac [!] d=0,
case-tac [!] e=0, case-tac [!] f=0, case-tac [!] g=0, case-tac [!] h=0)
by(simp-all add: *ipv6-preferred-to-compressed-pull-out-if max-zero-streak-def*)

lemma *ipv6-preferred-to-compressed*:
assumes *ipv6-unparsed-compressed-to-preferred* (*ipv6-preferred-to-compressed*
ip) = Some ip'
shows ip = ip'
proof —
from *assms have* 1: \exists *ipv6compressed*.
parse-ipv6-address-compressed (*ipv6-preferred-to-compressed* ip) = Some
ipv6compressed \wedge
ipv6addr-c2p ipv6compressed = ip' **using** *ipv6-unparsed-compressed-to-preferred-identity2*
by *simp*

obtain $a b c d e f g h$ **where** $ip: ip = IPv6AddrPreferred a b c d e f g h$ **by** (*cases ip*)

have *ipv6-preferred-to-compressed-None1*:

$ipv6\text{-preferred-to-compressed } (IPv6AddrPreferred a b c d e f g h) = None\#xs$
 \implies
 $(map\ Some\ (dropWhile\ (\lambda x. x=0))\ [a,b,c,d,e,f,g,h]) = xs \implies (IPv6AddrPreferred\ a\ b\ c\ d\ e\ f\ g\ h) = ip'$
 $(IPv6AddrPreferred\ a\ b\ c\ d\ e\ f\ g\ h) = ip'$ **for** xs
apply (*case-tac a=0, case-tac [!] b=0, case-tac [!] c=0, case-tac [!] d=0,*
case-tac [!] e=0, case-tac [!] f=0, case-tac [!] g=0, case-tac [!] h=0)
by (*simp-all add: ipv6-preferred-to-compressed-pull-out-if max-zero-streak-def*)

have *ipv6-preferred-to-compressed-None2*:

$ipv6\text{-preferred-to-compressed } (IPv6AddrPreferred a b c d e f g h) = (Some\ a')\#None\#xs \implies$
 $(map\ Some\ (dropWhile\ (\lambda x. x=0))\ [b,c,d,e,f,g,h]) = xs \implies (IPv6AddrPreferred\ a'\ b\ c\ d\ e\ f\ g\ h) = ip'$
 $(IPv6AddrPreferred\ a\ b\ c\ d\ e\ f\ g\ h) = ip'$ **for** $xs\ a'$
apply (*case-tac a=0, case-tac [!] b=0, case-tac [!] c=0, case-tac [!] d=0,*
case-tac [!] e=0, case-tac [!] f=0, case-tac [!] g=0, case-tac [!] h=0)
by (*simp-all add: ipv6-preferred-to-compressed-pull-out-if max-zero-streak-def*)

have *ipv6-preferred-to-compressed-None3*:

$ipv6\text{-preferred-to-compressed } (IPv6AddrPreferred a b c d e f g h) = (Some\ a')\#(Some\ b')\#None\#xs \implies$
 $(map\ Some\ (dropWhile\ (\lambda x. x=0))\ [c,d,e,f,g,h]) = xs \implies (IPv6AddrPreferred\ a'\ b'\ c\ d\ e\ f\ g\ h) = ip'$
 $(IPv6AddrPreferred\ a\ b\ c\ d\ e\ f\ g\ h) = ip'$ **for** $xs\ a'\ b'$
apply (*case-tac a=0, case-tac [!] b=0, case-tac [!] c=0, case-tac [!] d=0,*
case-tac [!] e=0, case-tac [!] f=0, case-tac [!] g=0, case-tac [!] h=0)
by (*simp-all add: ipv6-preferred-to-compressed-pull-out-if max-zero-streak-def*)

have *ipv6-preferred-to-compressed-None4*:

$ipv6\text{-preferred-to-compressed } (IPv6AddrPreferred a b c d e f g h) = (Some\ a')\#(Some\ b')\#(Some\ c')\#None\#xs \implies$
 $(map\ Some\ (dropWhile\ (\lambda x. x=0))\ [d,e,f,g,h]) = xs \implies (IPv6AddrPreferred\ a'\ b'\ c'\ d\ e\ f\ g\ h) = ip'$
 $(IPv6AddrPreferred\ a\ b\ c\ d\ e\ f\ g\ h) = ip'$ **for** $xs\ a'\ b'\ c'$
apply (*case-tac a=0, case-tac [!] b=0, case-tac [!] c=0, case-tac [!] d=0,*
case-tac [!] e=0, case-tac [!] f=0, case-tac [!] g=0, case-tac [!] h=0)
by (*simp-all add: ipv6-preferred-to-compressed-pull-out-if max-zero-streak-def*)

have *ipv6-preferred-to-compressed-None5*:

$ipv6\text{-preferred-to-compressed } (IPv6AddrPreferred a b c d e f g h) = (Some\ a')\#(Some\ b')\#(Some\ c')\#(Some\ d')\#None\#xs \implies$
 $(map\ Some\ (dropWhile\ (\lambda x. x=0))\ [e,f,g,h]) = xs \implies (IPv6AddrPreferred\ a'\ b'\ c'\ d'\ e\ f\ g\ h) = ip'$
 $(IPv6AddrPreferred\ a\ b\ c\ d\ e\ f\ g\ h) = ip'$ **for** $xs\ a'\ b'\ c'\ d'$

apply(*case-tac* *a=0,case-tac* [!] *b=0,case-tac* [!] *c=0,case-tac* [!] *d=0,*
case-tac [!] *e=0,case-tac* [!] *f=0,case-tac* [!] *g=0,case-tac* [!] *h=0*)
by(*simp-all add: ipv6-preferred-to-compressed-pull-out-if max-zero-streak-def*)

have *ipv6-preferred-to-compressed-None6*:

ipv6-preferred-to-compressed (IPv6AddrPreferred a b c d e f g h) = (Some a')#(Some b')#(Some c')#(Some d')#(Some e')#None#xs \implies
(map Some (dropWhile ($\lambda x. x=0$) [f,g,h]) = xs \implies *(IPv6AddrPreferred a' b' c' d' e' f' g' h) = ip'*) \implies
(IPv6AddrPreferred a b c d e f g h) = ip' **for** *xs a' b' c' d' e'*
apply(*case-tac* *a=0,case-tac* [!] *b=0,case-tac* [!] *c=0,case-tac* [!] *d=0,*
case-tac [!] *e=0,case-tac* [!] *f=0,case-tac* [!] *g=0,case-tac* [!] *h=0*)
by(*simp-all add: ipv6-preferred-to-compressed-pull-out-if max-zero-streak-def*)

have *ipv6-preferred-to-compressed-None7*:

ipv6-preferred-to-compressed (IPv6AddrPreferred a b c d e f g h) = (Some a')#(Some b')#(Some c')#(Some d')#(Some e')#(Some f')#None#xs \implies
(map Some (dropWhile ($\lambda x. x=0$) [g,h]) = xs \implies *(IPv6AddrPreferred a' b' c' d' e' f' g' h) = ip'*) \implies
(IPv6AddrPreferred a b c d e f g h) = ip' **for** *xs a' b' c' d' e' f'*
apply(*case-tac* *a=0,case-tac* [!] *b=0,case-tac* [!] *c=0,case-tac* [!] *d=0,*
case-tac [!] *e=0,case-tac* [!] *f=0,case-tac* [!] *g=0,case-tac* [!] *h=0*)
by(*simp-all add: ipv6-preferred-to-compressed-pull-out-if max-zero-streak-def*)

have *ipv6-preferred-to-compressed-None8*:

ipv6-preferred-to-compressed (IPv6AddrPreferred a b c d e f g h) = (Some a')#(Some b')#(Some c')#(Some d')#(Some e')#(Some f')#(Some g')#None#xs \implies
(map Some (dropWhile ($\lambda x. x=0$) [h]) = xs \implies *(IPv6AddrPreferred a' b' c' d' e' f' g' h) = ip'*) \implies
(IPv6AddrPreferred a b c d e f g h) = ip' **for** *xs a' b' c' d' e' f' g'*
apply(*case-tac* *a=0,case-tac* [!] *b=0,case-tac* [!] *c=0,case-tac* [!] *d=0,*
case-tac [!] *e=0,case-tac* [!] *f=0,case-tac* [!] *g=0,case-tac* [!] *h=0*)
by(*simp-all add: ipv6-preferred-to-compressed-pull-out-if max-zero-streak-def*)

have 2: *parse-ipv6-address-compressed (ipv6-preferred-to-compressed (IPv6AddrPreferred a b c d e f g h))*

= Some ipv6compressed \implies

ipv6addr-c2p ipv6compressed = ip' \implies

IPv6AddrPreferred a b c d e f g h = ip'

for *ipv6compressed*

apply(*erule parse-ipv6-address-compressed-someE*)

apply(*simp-all*)

apply(*erule ipv6-preferred-to-compressed-None1,*

simp split: if-split-asm)+

apply(*erule ipv6-preferred-to-compressed-None2, simp*

split: if-split-asm)+

apply(*erule ipv6-preferred-to-compressed-None3, simp split:*

```

if-split-asm)+
      apply(erule ipv6-preferred-to-compressed-None4, simp split:
if-split-asm)+
      apply(erule ipv6-preferred-to-compressed-None5, simp split: if-split-asm)+
      apply(erule ipv6-preferred-to-compressed-None6, simp split: if-split-asm)+
      apply(erule ipv6-preferred-to-compressed-None7, simp split: if-split-asm)+
      apply(erule ipv6-preferred-to-compressed-None8, simp split: if-split-asm)
    done
  from 1 2 ip show ?thesis by(elim exE conjE, simp)
qed

end

end

theory Prefix-Match
imports IP-Address
begin

```

6 Prefix Match

The main difference between the prefix match defined here and CIDR notation is a validity constraint imposed on prefix matches.

For example, 192.168.42.42/16 is valid CIDR notation whereas for a prefix match, it must be 192.168.0.0/16.

I.e. the last bits of the prefix must be set to zero.

```

context
  notes [[typedef-overloaded]]
begin
  datatype 'a prefix-match = PrefixMatch (pfm-prefix: 'a::len word) (pfm-length:
nat)
end

```

```

definition pfm-mask :: 'a prefix-match  $\Rightarrow$  'a::len word where
  pfm-mask x  $\equiv$  mask (len-of (TYPE('a)) - pfm-length x)

```

```

context
  includes bit-operations-syntax
begin

```

```

definition valid-prefix :: ('a::len) prefix-match  $\Rightarrow$  bool where
  valid-prefix pf = ((pfm-mask pf) AND pfm-prefix pf = 0)

```

Note that *valid-prefix* looks very elegant as a definition. However, it hides something nasty:

```

lemma valid-prefix (PrefixMatch (0::32 word) 42) by eval

```

When zeroing all least significant bits which exceed the *pf_xm-length*, you get a *valid-prefix*

lemma *mk-valid-prefix*:

fixes *base::'a::len word*

shows *valid-prefix (PrefixMatch (base AND NOT (mask (len-of TYPE ('a) – len))) len)*

proof –

have *mask (len – m) AND base AND NOT (mask (len – m)) = 0*

for *m len and base::'a::len word* **by** (*simp add: word-bw-lcs*)

thus *?thesis*

by (*simp add: valid-prefix-def pf_xm-mask-def pf_xm-length-def pf_xm-prefix-def*)

qed

end

The type *'a prefix-match* usually requires *valid-prefix*. When we allow working on arbitrary IPs in CIDR notation, we will use the type *'i word × nat* directly.

lemma *valid-prefix-00: valid-prefix (PrefixMatch 0 0) by (simp add: valid-prefix-def)*

definition *prefix-match-to-CIDR :: ('i::len) prefix-match ⇒ ('i word × nat) where*
prefix-match-to-CIDR pfx ≡ (pf_xm-prefix pfx, pf_xm-length pfx)

lemma *prefix-match-to-CIDR-def2: prefix-match-to-CIDR = (λpfx. (pf_xm-prefix pfx, pf_xm-length pfx))*

unfolding *prefix-match-to-CIDR-def fun-eq-iff* **by** *simp*

definition *prefix-match-dtor m ≡ (case m of PrefixMatch p l ⇒ (p,l))*

Some more or less random linear order on prefixes. Only used for serialization at the time of this writing.

instantiation *prefix-match :: (len) linorder*

begin

definition *a ≤ b ↔ (if pf_xm-length a = pf_xm-length b*
then pf_xm-prefix a ≤ pf_xm-prefix b
else pf_xm-length a > pf_xm-length b)

definition *a < b ↔ (a ≠ b ∧*
(if pf_xm-length a = pf_xm-length b
then pf_xm-prefix a ≤ pf_xm-prefix b
else pf_xm-length a > pf_xm-length b))

instance

by *standard (auto simp: less-eq-prefix-match-def less-prefix-match-def prefix-match.expand split: if-splits)*

end

lemma *sorted-list-of-set*

{PrefixMatch 0 32 :: 32 prefix-match,

```

    PrefixMatch 42 32,
    PrefixMatch 0 0,
    PrefixMatch 0 1,
    PrefixMatch 12 31 } =
  [PrefixMatch 0 32, PrefixMatch 0x2A 32, PrefixMatch 0xC 31, PrefixMatch 0
  1, PrefixMatch 0 0]
  by eval

```

context

```

  includes bit-operations-syntax
begin

```

private lemma *valid-prefix-E*: $\text{valid-prefix } pf \implies ((\text{pfxm-mask } pf) \text{ AND } \text{pfxm-prefix } pf = 0)$

unfolding *valid-prefix-def* .

private lemma *valid-prefix-alt*: **fixes** $p::'a::\text{len } \text{prefix-match}$

shows $\text{valid-prefix } p = (\text{pfxm-prefix } p \text{ AND } (2^{\wedge}((\text{len-of TYPE } ('a)) - \text{pfxm-length } p) - 1) = 0)$

unfolding *valid-prefix-def*

unfolding *mask-eq*

using *word-bw-comms(1)*

arg-cong[**where** $f = \lambda x. (\text{pfxm-prefix } p \text{ AND } x - 1 = 0)$]

shiffl-1

unfolding *pfxm-prefix-def pfxm-mask-def mask-eq*

apply (*cases p*)

apply (*simp add: ac-simps push-bit-of-1*)

done

6.1 Address Semantics

Matching on a $'a$ *prefix-match*. Think of routing tables.

definition *prefix-match-semantics* **where**

$\text{prefix-match-semantics } m a \equiv \text{pfxm-prefix } m = \text{NOT } (\text{pfxm-mask } m) \text{ AND } a$

lemma *same-length-prefixes-distinct*: $\text{valid-prefix } pfx1 \implies \text{valid-prefix } pfx2 \implies pfx1 \neq pfx2 \implies \text{pfxm-length } pfx1 = \text{pfxm-length } pfx2 \implies \text{prefix-match-semantics } pfx1 w \implies \text{prefix-match-semantics } pfx2 w \implies \text{False}$

by (*simp add: pfxm-mask-def prefix-match.expand prefix-match-semantics-def*)

6.2 Relation between prefix and set

definition *prefix-to-wordset* $:: 'a::\text{len } \text{prefix-match} \Rightarrow 'a \text{ word set}$ **where**

$\text{prefix-to-wordset } pfx = \{\text{pfxm-prefix } pfx .. \text{pfxm-prefix } pfx \text{ OR } \text{pfxm-mask } pfx\}$

private lemma *pfx-not-empty*: $\text{valid-prefix } pfx \implies \text{prefix-to-wordset } pfx \neq \{\}$

unfolding *valid-prefix-def prefix-to-wordset-def* **by** (*simp add: le-word-or2*)

lemma *zero-prefix-match-all*:

$\text{valid-prefix } m \implies \text{pfxm-length } m = 0 \implies \text{prefix-match-semantics } m ip$

by(*simp add: pfxm-mask-def mask-2pm1 valid-prefix-alt prefix-match-semantics-def*)

lemma *prefix-to-wordset-subset-ipset-from-cidr*:

prefix-to-wordset pfx \subseteq *ipset-from-cidr (pfxm-prefix pfx) (pfxm-length pfx)*

apply(*rule subsetI*)

apply(*simp add: prefix-to-wordset-def addr-in-ipset-from-cidr-code*)

apply(*intro impI conjI*)

apply (*metis (erased, opaque-lifting) order-trans word-and-le2*)

apply(*simp add: pfxm-mask-def*)

done

6.3 Equivalence Proofs

theorem *prefix-match-semantics-wordset*:

assumes *valid-prefix pfx*

shows *prefix-match-semantics pfx a* \longleftrightarrow $a \in$ *prefix-to-wordset pfx*

using *assms*

unfolding *valid-prefix-def pfxm-mask-def prefix-match-semantics-def prefix-to-wordset-def*

apply(*cases pfx, rename-tac base len*)

apply(*simp*)

apply(*drule-tac base=base and len=len and a=a in zero-base-lsb-imp-set-eq-as-bit-operation*)

by (*simp*)

private lemma *valid-prefix-ipset-from-netmask-ipset-from-cidr*:

shows *ipset-from-netmask (pfxm-prefix pfx) (NOT (pfxm-mask pfx)) =*
ipset-from-cidr (pfxm-prefix pfx) (pfxm-length pfx)

apply(*cases pfx*)

apply(*simp add: ipset-from-cidr-alt2 pfxm-mask-def*)

done

lemma *prefix-match-semantics-ipset-from-netmask*:

assumes *valid-prefix pfx*

shows *prefix-match-semantics pfx a* \longleftrightarrow

$a \in$ *ipset-from-netmask (pfxm-prefix pfx) (NOT (pfxm-mask pfx))*

unfolding *prefix-match-semantics-wordset[OF assms]*

unfolding *valid-prefix-ipset-from-netmask-ipset-from-cidr*

unfolding *prefix-to-wordset-def*

apply(*subst ipset-from-cidr-base-wellforemd*)

subgoal using *assms by (simp add: valid-prefix-def pfxm-mask-def)*

by(*simp add: pfxm-mask-def*)

lemma *prefix-match-semantics-ipset-from-netmask2*:

assumes *valid-prefix pfx*

shows *prefix-match-semantics pfx (a :: 'i::len word)* \longleftrightarrow

$a \in$ *ipset-from-cidr (pfxm-prefix pfx) (pfxm-length pfx)*

unfolding *prefix-match-semantics-ipset-from-netmask[OF assms] pfxm-mask-def*
ipset-from-cidr-def

by (*metis (full-types) NOT-mask-shifted-lenword word-not-not*)

```

lemma prefix-to-wordset-ipset-from-cidr:
  assumes valid-prefix (px::'a::len prefix-match)
  shows prefix-to-wordset px = ipset-from-cidr (pxm-prefix px) (pxm-length
px)
proof –
  have helper3: (x::'a::len word) OR y = x OR y AND NOT x for x y by (simp
add: word-oa-dist2)
  have prefix-match-semantics-ipset-from-netmask:
    (prefix-to-wordset px) = ipset-from-netmask (pxm-prefix px) (NOT
(pxm-mask px))
  unfolding prefix-to-wordset-def ipset-from-netmask-def Let-def
  using assms
  by (clarsimp dest!: valid-prefix-E) (metis bit.conj-commute mask-eq-0-eq-x)
  have ((mask len)::'a::len word) << LENGTH('a) – len = ~ (mask (LENGTH('a)
– len))
  for len using NOT-mask-shifted-lenword by (metis word-not-not)
  from this[of (pxm-length px)] have mask-def2-symmetric:
    ((mask (pxm-length px)::'a::len word) << LENGTH('a) – pxm-length px)
=
  NOT (pxm-mask px)
  unfolding pxm-mask-def by simp

have ipset-from-netmask-prefix:
  ipset-from-netmask (pxm-prefix px) (NOT (pxm-mask px)) =
  ipset-from-cidr (pxm-prefix px) (pxm-length px)
  unfolding ipset-from-netmask-def ipset-from-cidr-alt
  unfolding pxm-mask-def[symmetric]
  unfolding mask-def2-symmetric
  apply(simp)
  unfolding Let-def
  using assms[unfolded valid-prefix-def]
  by (metis helper3 word-bw-comms(2))

show ?thesis by (metis ipset-from-netmask-prefix local.prefix-match-semantics-ipset-from-netmask)

qed

definition prefix-to-wordinterval :: 'a::len prefix-match ⇒ 'a wordinterval where
  prefix-to-wordinterval px ≡ WordInterval (pxm-prefix px) (pxm-prefix px OR
pxm-mask px)

lemma prefix-to-wordinterval-set-eq[simp]:
  wordinterval-to-set (prefix-to-wordinterval px) = prefix-to-wordset px
  unfolding prefix-to-wordinterval-def prefix-to-wordset-def by simp

lemma prefix-to-wordinterval-def2:
  prefix-to-wordinterval px =
  iprange-interval ((pxm-prefix px), (pxm-prefix px OR pxm-mask px))

```

```

unfolding iprange-interval.simps prefix-to-wordinterval-def by simp
corollary prefix-to-wordinterval-ipset-from-cidr: valid-prefix pfx  $\implies$ 
  wordinterval-to-set (prefix-to-wordinterval pfx) =
  ipset-from-cidr (pfxm-prefix pfx) (pfxm-length pfx)
using prefix-to-wordset-ipset-from-cidr prefix-to-wordinterval-set-eq by auto
end

```

```

lemma prefix-never-empty:
  fixes d:: 'a::len prefix-match
  shows  $\neg$  wordinterval-empty (prefix-to-wordinterval d)
by (simp add: le-word-or2 prefix-to-wordinterval-def)

```

Getting a lowest element

```

lemma ipset-from-cidr-lowest: a  $\in$  ipset-from-cidr a n
  using ip-cidr-set-def ipset-from-cidr-eq-ip-cidr-set by blast

```

```

lemma valid-prefix (PrefixMatch a n)  $\implies$  is-lowest-element a (ipset-from-cidr a n)
  apply (simp add: is-lowest-element-def ipset-from-cidr-lowest)
  apply (simp add: ipset-from-cidr-eq-ip-cidr-set ip-cidr-set-def)
  apply (simp add: valid-prefix-def pfxm-mask-def)
  by (metis diff-zero eq-iff mask-out-sub-mask word-and-le2 word-bw-comms(1))

```

end

```

theory CIDR-Split
imports IP-Address
  Prefix-Match
  Hs-Compat
begin

```

7 CIDR Split Motivation (Example for IPv4)

When talking about ranges of IP addresses, we can make the ranges explicit by listing their elements.

```

context
begin

```

```

  private lemma map (of-nat  $\circ$  nat) [1 .. 4] = ([1, 2, 3, 4]:: 32 word list) by
  eval

```

```

  private definition ipv4addr-upto :: 32 word  $\Rightarrow$  32 word  $\Rightarrow$  32 word list where
  ipv4addr-upto i j  $\equiv$  map (of-nat  $\circ$  nat) [int (unat i) .. int (unat j)]

```

```

  private lemma ipv4addr-upto: set (ipv4addr-upto i j) = {i .. j}

```

proof –

```

  have int-interval-eq-image: {int m..int n} = int ‘ {m..n} for m n
  by (auto intro!: image-eqI [of - int nat k for k])

```

```

  have helpX: $\bigwedge$ f (i::nat) (j::nat). (f  $\circ$  nat) ‘ {int i..int j} = f ‘ {i .. j}

```

```

    by (auto simp add: image-comp int-interval-eq-image)
  have hlp: ⟨i ≤ word-of-nat (nat xa)⟩ ⟨word-of-nat (nat xa) ≤ j⟩
    if ⟨uint i ≤ xa⟩ ⟨xa ≤ uint j⟩ for xa :: int
  proof -
    from uint-nonnegative [of i] ⟨uint i ≤ xa⟩
      have ⟨0 ≤ xa⟩ by (rule order-trans)
    moreover from ⟨xa ≤ uint j⟩ uint-bounded [of j]
      have ⟨xa < 2 ^ 32⟩ by simp
    ultimately have xa: ⟨take-bit 32 xa = xa⟩
      by (simp add: take-bit-int-eq-self)
    from xa ⟨uint i ≤ xa⟩ show ⟨i ≤ word-of-nat (nat xa)⟩
      by transfer simp
    from xa ⟨xa ≤ uint j⟩ show ⟨word-of-nat (nat xa) ≤ j⟩
      by transfer simp
  qed
show ?thesis
  unfolding ipv4addr-upto-def
  apply (rule set-eqI)
  apply (auto simp add: hlp)
  apply (metis (mono-tags) atLeastAtMost-iff image-iff unat-eq-nat-uint word-less-eq-iff-unsigned
word-unat.Rep-inverse)
  done
qed

```

The function `ipv4addr-upto` gives back a list of all the ips in the list. This list can be pretty huge! In the following, we will use CIDR notation (e.g. `192.168.0.0/24`) to describe the list more compactly.

end

8 CIDR Split

```

context
begin

```

```

private lemma find-SomeD: find f x = Some y ⟹ f y ∧ y ∈ set x
  by (induction x; simp split: if-splits)

```

```

private definition pfxes :: 'a::len0 itself ⇒ nat list where
  pfxes = map nat [0..int(len-of TYPE ('a))]
private lemma pfxes TYPE(32) = map nat [0 .. 32] by eval

```

```

private definition largest-contained-prefix (a::('a :: len) word) r = (
  let cs = (map (λs. PrefixMatch a s) (pfxes TYPE('a)));
    — anything that is a subset should also be a valid prefix. but try proving that.
    cfs = find (λs. valid-prefix s ∧ wordinterval-subset (prefix-to-wordinterval s)
r) cs in
  cfs)

```

Split off one prefix:

```

private definition wordinterval-CIDR-split1
  :: 'a::len wordinterval  $\Rightarrow$  'a prefix-match option  $\times$  'a wordinterval where
  wordinterval-CIDR-split1 r  $\equiv$  (
    let ma = wordinterval-lowest-element r in
    case ma of
      None  $\Rightarrow$  (None, r) |
      Some a  $\Rightarrow$  (case largest-contained-prefix a r of
        None  $\Rightarrow$  (None, r) |
        Some m  $\Rightarrow$  (Some m, wordinterval-setminus r (prefix-to-wordinterval m))))

private lemma wordinterval-CIDR-split1-innard-helper: fixes a::'a::len word
  shows wordinterval-lowest-element r = Some a  $\implies$ 
  largest-contained-prefix a r  $\neq$  None
proof -
  assume a: wordinterval-lowest-element r = Some a
  have b: (a, len-of (TYPE('a)))  $\in$  set (map (Pair a) (pfxes TYPE('a)))
  unfolding pfxes-def set-map set-upto
  using Set.image-iff atLeastAtMost-iff int-eq-iff order-refl by metis
  have c: valid-prefix (PrefixMatch a (len-of (TYPE('a)))) by (simp add: valid-prefix-def
  pfxm-mask-def)
  have wordinterval-to-set (prefix-to-wordinterval (PrefixMatch a (len-of (TYPE('a')))))
  = {a}
  unfolding prefix-to-wordinterval-def pfxm-mask-def by simp
  moreover have a  $\in$  wordinterval-to-set r
  using a wordinterval-lowest-element-set-eq wordinterval-lowest-none-empty
  by (metis is-lowest-element-def option.distinct(1))
  ultimately have d:
  wordinterval-to-set (prefix-to-wordinterval (PrefixMatch a (LENGTH('a'))))  $\subseteq$ 
  wordinterval-to-set r
  by simp
  show ?thesis
  unfolding largest-contained-prefix-def Let-def
  using b c d by (auto simp add: find-None-iff)
qed

private lemma r-split1-not-none: fixes r:: 'a::len wordinterval
  shows  $\neg$  wordinterval-empty r  $\implies$  fst (wordinterval-CIDR-split1 r)  $\neq$  None
  unfolding wordinterval-CIDR-split1-def Let-def
  by (cases wordinterval-lowest-element r)
  (auto simp add: wordinterval-lowest-none-empty
  dest: wordinterval-CIDR-split1-innard-helper)

private lemma largest-contained-prefix-subset:
  largest-contained-prefix a r = Some p  $\implies$  wordinterval-to-set (prefix-to-wordinterval
  p)  $\subseteq$  wordinterval-to-set r
  unfolding largest-contained-prefix-def Let-def

```

by(*drule find-SomeD*) *simp*

private lemma *wordinterval-CIDR-split1-snd*: *wordinterval-CIDR-split1* $r = (\text{Some } s, u) \implies u = \text{wordinterval-setminus } r (\text{prefix-to-wordinterval } s)$
unfolding *wordinterval-CIDR-split1-def* *Let-def* **by**(*clarsimp split: option.splits*)

private lemma *largest-contained-prefix-subset-s1D*:
wordinterval-CIDR-split1 $r = (\text{Some } s, u) \implies \text{wordinterval-to-set } (\text{prefix-to-wordinterval } s) \subseteq \text{wordinterval-to-set } r$
by(*intro largest-contained-prefix-subset*[**where** $a = \text{the } (\text{wordinterval-lowest-element } r)$])
(simp add: wordinterval-CIDR-split1-def split: option.splits)

private theorem *wordinterval-CIDR-split1-preserve*: **fixes** $r:: 'a::\text{len wordinterval}$
shows *wordinterval-CIDR-split1* $r = (\text{Some } s, u) \implies \text{wordinterval-eq } (\text{wordinterval-union } (\text{prefix-to-wordinterval } s) u) r$
proof(*unfold wordinterval-eq-set-eq*)
assume $as: \text{wordinterval-CIDR-split1 } r = (\text{Some } s, u)$
have $ud: u = \text{wordinterval-setminus } r (\text{prefix-to-wordinterval } s)$
using as [*THEN wordinterval-CIDR-split1-snd*] .
with *largest-contained-prefix-subset-s1D*[*OF as*]
show *wordinterval-to-set* (*wordinterval-union* (*prefix-to-wordinterval* s) u) = *wordinterval-to-set* r
unfolding ud **by** *auto*
qed

private lemma *wordinterval-CIDR-split1-some-r-ne*:
wordinterval-CIDR-split1 $r = (\text{Some } s, u) \implies \neg \text{wordinterval-empty } r$
proof(*rule ccontr, goal-cases*)
case 1
have *wordinterval-lowest-element* $r = \text{None}$ **unfolding** *wordinterval-lowest-none-empty*
using 1(2) **unfolding** *not-not* .
then have *wordinterval-CIDR-split1* $r = (\text{None}, r)$ **unfolding** *wordinterval-CIDR-split1-def*
Let-def **by** *simp*
then show *False* **using** 1(1) **by** *simp*
qed

private lemma *wordinterval-CIDR-split1-distinct*: **fixes** $r:: 'a::\text{len wordinterval}$
shows *wordinterval-CIDR-split1* $r = (\text{Some } s, u) \implies \text{wordinterval-empty } (\text{wordinterval-intersection } (\text{prefix-to-wordinterval } s) u)$
proof(*goal-cases*)
case 1
have $nn: \text{wordinterval-lowest-element } r \neq \text{None}$
using *wordinterval-CIDR-split1-some-r-ne* 1 *wordinterval-lowest-none-empty*
by *metis*
from 1 **have** $u = \text{wordinterval-setminus } r (\text{prefix-to-wordinterval } s)$
by(*elim wordinterval-CIDR-split1-snd*)
then show *?thesis* **by** *simp*
qed

private lemma *wordinterval-CIDR-split1-distinct2*: **fixes** *r*:: 'a::len *wordinterval*
shows *wordinterval-CIDR-split1* *r* = (Some *s*, *u*) \implies
wordinterval-empty (*wordinterval-intersection* (*prefix-to-wordinterval* *s*) *u*)
by(*rule wordinterval-CIDR-split1-distinct*[**where** *r* = *r*]) *simp*

function *wordinterval-CIDR-split-prefixmatch*
:: 'a::len *wordinterval* \Rightarrow 'a *prefix-match list* **where**
wordinterval-CIDR-split-prefixmatch *rs* = (
 if
 \neg *wordinterval-empty* *rs*
 then *case wordinterval-CIDR-split1* *rs*
 of (Some *s*, *u*) \Rightarrow *s* # *wordinterval-CIDR-split-prefixmatch* *u*
 | - \Rightarrow []
 else
 []
)
by *pat-completeness simp*

termination *wordinterval-CIDR-split-prefixmatch*
proof(*relation measure* (*card* \circ *wordinterval-to-set*), *rule wf-measure*, *unfold in-measure*
comp-def, *goal-cases*)
note *vernichter* = *wordinterval-empty-set-eq wordinterval-intersection-set-eq wordinter-*
val-union-set-eq wordinterval-eq-set-eq
case (1 *rs* *x* *y* *x2*)
note *some* = 1(2)[*unfolded* 1(3), *symmetric*]
from *prefix-never-empty* **have** *wordinterval-to-set* (*prefix-to-wordinterval* *x2*) \neq
{} **unfolding** *vernichter* .
thus ?*case*
 unfolding *wordinterval-CIDR-split1-preserve*[*OF some*, *unfolded vernichter*,
symmetric]
 unfolding *card-Un-disjoint*[*OF finite finite wordinterval-CIDR-split1-distinct*[*OF*
some, *unfolded vernichter*]]
 by *auto*
qed

private lemma *unfold-rsplit-case*:
assumes *su*: (Some *s*, *u*) = *wordinterval-CIDR-split1* *rs*
shows (*case wordinterval-CIDR-split1* *rs* of (None, *u*) \Rightarrow []
 | (Some *s*, *u*) \Rightarrow *s* # *wordinterval-CIDR-split-prefixmatch*
u) = *s* # *wordinterval-CIDR-split-prefixmatch* *u*
using *su* **by** (*metis option.simps*(5) *split-conv*)

lemma *wordinterval-CIDR-split-prefixmatch*
(*RangeUnion* (*WordInterval* (0x40000000) 0x5FEFBBCC) (*WordInterval*
0x5FEEBB1C 0x7FFFFFFF))
= [*PrefixMatch* (0x40000000::32 *word*) 2] **by** *eval*

lemma *length* (*wordinterval-CIDR-split-prefixmatch* (*WordInterval* 0 (0xFFFFFFFFE::32
word))) = 32 **by** *eval*

declare *wordinterval-CIDR-split-prefixmatch.simps*[*simp del*]

theorem *wordinterval-CIDR-split-prefixmatch*:

wordinterval-to-set $r = (\bigcup x \in \text{set} (\text{wordinterval-CIDR-split-prefixmatch } r)). \text{prefix-to-wordset } x$

proof(*induction* r *rule*: *wordinterval-CIDR-split-prefixmatch.induct*)

case (1 rs)

show $?case$ **proof**(*cases* *wordinterval-empty* rs)

case *True*

thus $?thesis$ **by**(*simp add*: *wordinterval-CIDR-split-prefixmatch.simps*)

next

case *False*

obtain x y **where** $s1$: *wordinterval-CIDR-split1* $rs = (\text{Some } x, y)$

using *r-split1-not-none*[*OF False*] **by**(*auto simp add*: *fst-def split: prod.splits*)

have mIH : *wordinterval-to-set* $y = (\bigcup x \in \text{set} (\text{wordinterval-CIDR-split-prefixmatch } y)). \text{prefix-to-wordset } x$

using 1 [*OF False s1*[*symmetric*] *refl*] .

have $*$: *wordinterval-to-set* $rs = \text{prefix-to-wordset } x \cup (\bigcup x \in \text{set} (\text{wordinterval-CIDR-split-prefixmatch } y)). \text{prefix-to-wordset } x$

unfolding mIH [*symmetric*]

proof –

have ud : $y = \text{wordinterval-setminus } rs (\text{prefix-to-wordinterval } x)$

using *wordinterval-CIDR-split1-snd*[*OF s1*] .

have ss : $\text{prefix-to-wordset } x \subseteq \text{wordinterval-to-set } rs$

using *largest-contained-prefix-subset-s1D*[*OF s1*] **by** *simp*

show *wordinterval-to-set* $rs = \text{prefix-to-wordset } x \cup \text{wordinterval-to-set } y$

unfolding ud **using** ss **by** *simp blast*

qed

show $?thesis$

apply(*subst* *wordinterval-CIDR-split-prefixmatch.simps*)

apply(*unfold if-P*[*OF False*] $s1$ *prod.simps option.simps* $*$)

apply(*simp*)

done

qed

qed

lemma *wordinterval-CIDR-split-prefixmatch-all-valid-Ball*: **fixes** r :: $'a$::*len* *wordinterval*

shows $\forall e \in \text{set} (\text{wordinterval-CIDR-split-prefixmatch } r). \text{valid-prefix } e \wedge \text{pfm-length } e \leq \text{LENGTH}('a)$

proof(*induction* r *rule*: *wordinterval-CIDR-split-prefixmatch.induct*)

case 1

case (1 rs)

show $?case$ **proof**(*cases* *wordinterval-empty* rs)

case *False*

obtain x y **where** $s1$: *wordinterval-CIDR-split1* $rs = (\text{Some } x, y)$

```

    using r-split1-not-none[OF False] by(auto simp add: fst-def split: prod.splits)
  hence i1: valid-prefix x
    unfolding wordinterval-CIDR-split1-def Let-def largest-contained-prefix-def
    by(auto dest: find-SomeD split: option.splits)
  have i2: pfxm-length x ≤ LENGTH('a)
    using s1 unfolding wordinterval-CIDR-split1-def Let-def largest-contained-prefix-def
  pfxes-def
    by(force split: option.splits dest: find-SomeD simp: nat-le-iff)
  have mIH: ∀ a∈set (wordinterval-CIDR-split-prefixmatch y). valid-prefix a ∧
  pfxm-length a ≤ LENGTH('a)
    using 1[OF False s1[symmetric] refl] .
  with i1 i2 show ?thesis
    apply(subst wordinterval-CIDR-split-prefixmatch.simps)
    apply(unfold if-P[OF False] s1 prod.simps option.simps)
    apply(simp)
  done
qed (simp add: wordinterval-CIDR-split-prefixmatch.simps)
qed

```

```

private lemma wordinterval-CIDR-split-prefixmatch-all-valid-less-Ball-hlp:
  x ∈ set [s←map (PrefixMatch x2) (pfxes TYPE('a::len0))] . valid-prefix s ∧ wordinter-
  val-to-set (prefix-to-wordinterval s) ⊆ wordinterval-to-set rs] ⇒ pfxm-length x ≤
  LENGTH('a)
by(clarsimp simp: pfxes-def) presburger

```

Since `wordinterval-CIDR-split-prefixmatch` only returns valid prefixes, we can safely convert it to CIDR lists

```

lemma valid-prefix (PrefixMatch (0::16 word) 20) by(simp add: valid-prefix-def)

```

```

lemma wordinterval-CIDR-split-disjunct: a ∈ set (wordinterval-CIDR-split-prefixmatch
i) ⇒

```

```

  b ∈ set (wordinterval-CIDR-split-prefixmatch i) ⇒ a ≠ b ⇒
  prefix-to-wordset a ∩ prefix-to-wordset b = {}

```

```

proof(induction i rule: wordinterval-CIDR-split-prefixmatch.induct)

```

```

  case (1 rs)

```

```

  note IH = 1(1)

```

```

  have prema: a ∈ set (wordinterval-CIDR-split-prefixmatch rs) (is a ∈ ?os) using
  1 by simp

```

```

  have premb: b ∈ ?os using 1 by simp

```

```

  show ?case proof(cases wordinterval-empty rs)

```

```

    case False

```

```

    obtain x y where s1: wordinterval-CIDR-split1 rs = (Some x, y)

```

```

    using r-split1-not-none[OF False] by(auto simp add: fst-def split: prod.splits)

```

```

    have mi: k ∈ set (wordinterval-CIDR-split-prefixmatch y) (is k ∈ ?rs)

```

```

    if p: k ≠ x k ∈ ?os for k using p s1

```

```

    by(subst (asm) wordinterval-CIDR-split-prefixmatch.simps) (simp only: if-P[OF
False] split: prod.splits option.splits; simp)

```

```

    have a: k ∈ ?rs ⇒ prefix-to-wordset k ⊆ wordinterval-to-set y for k

```

```

unfolding wordinterval-CIDR-split-prefixmatch by blast
have b: prefix-to-wordset  $x \cap \text{wordinterval-to-set } y = \{\}$ 
using wordinterval-CIDR-split1-snd[OF s1] by simp
show ?thesis
proof(cases a = x; cases b = x)
  assume as:  $a = x \ b \neq x$ 
  with a[OF mi[OF as(2) premb]] b
  show ?thesis by blast
next
  assume as:  $a \neq x \ b = x$ 
  with a[OF mi[OF as(1) prema]] b
  show ?thesis by blast
next
  assume as:  $a \neq x \ b \neq x$ 
  have i:  $a \in ?rs \ b \in ?rs$ 
  using as mi prema premb by blast+
  show prefix-to-wordset a  $\cap$  prefix-to-wordset b =  $\{\}$ 
  by(rule IH[OF False s1[symmetric] refl i]) (fact 1)
next
  assume as:  $a = x \ b = x$ 
  with 1 have False by simp
  thus ?thesis ..
qed
next
  case True
  hence wordinterval-CIDR-split-prefixmatch rs =  $\square$  by(simp add: wordinterval-CIDR-split-prefixmatch.simps)
  thus ?thesis using prema by simp
qed
qed

```

lemma *wordinterval-CIDR-split-distinct*: *distinct (wordinterval-CIDR-split-prefixmatch i)*

```

proof(induction i rule: wordinterval-CIDR-split-prefixmatch.induct)
case (1 rs)
show ?case proof(cases wordinterval-empty rs)
  case False
  obtain x y where s1: wordinterval-CIDR-split1 rs = (Some x, y)
  using r-split1-not-none[OF False] by(auto simp add: fst-def split: prod.splits)
  have mIH: distinct (wordinterval-CIDR-split-prefixmatch y)
  using 1[OF False s1[symmetric] refl] .
  have prefix-to-wordset x  $\cap$  wordinterval-to-set y =  $\{\}$ 
  using wordinterval-CIDR-split1-snd[OF s1] by simp
  hence i1:  $x \notin \text{set } (\text{wordinterval-CIDR-split-prefixmatch } y)$ 
  unfolding wordinterval-CIDR-split-prefixmatch using prefix-never-empty[of
x, simplified] by blast
  show ?thesis

```

```

using s1
by (subst wordinterval-CIDR-split-prefixmatch.simps)
    (simp add: if-P[OF False] mIH i1 split: option.splits prod.splits)
qed (simp add: wordinterval-CIDR-split-prefixmatch.simps)
qed

```

lemma *wordinterval-CIDR-split-existential*:
 $x \in \text{wordinterval-to-set } w \implies \exists s. s \in \text{set } (\text{wordinterval-CIDR-split-prefixmatch } w) \wedge x \in \text{prefix-to-wordset } s$
using *wordinterval-CIDR-split-prefixmatch[symmetric]* **by** *fastforce*

8.1 Versions for *ipset-from-cidr*

definition *cidr-split* :: '*i*::len wordinterval \Rightarrow ('*i* word \times nat) list **where**
cidr-split *rs* \equiv *map prefix-match-to-CIDR (wordinterval-CIDR-split-prefixmatch rs)*

corollary *cidr-split-prefix*:
fixes *r* :: '*i*::len wordinterval
shows $(\bigcup x \in \text{set } (\text{cidr-split } r). \text{uncurry } \text{ipset-from-cidr } x) = \text{wordinterval-to-set } r$
unfolding *wordinterval-CIDR-split-prefixmatch[symmetric]* *cidr-split-def*
apply (simp add: *prefix-match-to-CIDR-def2* *wordinterval-CIDR-split-prefixmatch*)
using *prefix-to-wordset-ipset-from-cidr* *wordinterval-CIDR-split-prefixmatch-all-valid-Ball*
by *blast*

corollary *cidr-split-prefix-single*:
fixes *start* :: '*i*::len word
shows $(\bigcup x \in \text{set } (\text{cidr-split } (\text{iprange-interval } (\text{start}, \text{end}))). \text{uncurry } \text{ipset-from-cidr } x) = \{\text{start}.. \text{end}\}$
unfolding *wordinterval-to-set.simps[symmetric]*
using *cidr-split-prefix iprange-interval.simps* **by** *metis*

private lemma *interval-in-splitD*: $xa \in \text{foo} \implies \text{prefix-to-wordset } xa \subseteq \bigcup (\text{prefix-to-wordset } ' \text{foo})$ **by** *auto*

lemma *cidrsplit-no-overlaps*: \llbracket
 $x \in \text{set } (\text{wordinterval-CIDR-split-prefixmatch } wi);$
 $xa \in \text{set } (\text{wordinterval-CIDR-split-prefixmatch } wi);$
 $pt \ \&\& \ \sim\sim \ (\text{pfxm-mask } x) = \text{pfxm-prefix } x;$
 $pt \ \&\& \ \sim\sim \ (\text{pfxm-mask } xa) = \text{pfxm-prefix } xa$
 \rrbracket
 $\implies x = xa$

proof (rule *ccontr*, *goal-cases*)

case 1

hence *prefix-match-semantics* *x* *pt* *prefix-match-semantics* *xa* *pt* **unfolding** *prefix-match-semantics-def* **by** (simp-all add: *word-bw-comms(1)*)

moreover **have** *valid-prefix* *x* *valid-prefix* *xa* **using** 1 (1–2) *wordinterval-CIDR-split-prefixmatch-all-valid-Ball*
by *blast+*

ultimately **have** $pt \in \text{prefix-to-wordset } x$ $pt \in \text{prefix-to-wordset } xa$

```

using prefix-match-semantics-wordset by blast+
with wordinterval-CIDR-split-disjunct[OF 1(1,2) 1(5)] show False by blast
qed

end

```

```

end
theory WordInterval-Sorted
imports WordInterval
         Automatic-Refinement.Misc
         HOL-Library.Product-Lexorder
begin

```

Use this and *wordinterval-to-set* (*wordinterval-compress* ?r) = *wordinterval-to-set* ?r before pretty-printing.

```

definition wordinterval-sort :: 'a::len wordinterval  $\Rightarrow$  'a::len wordinterval where
  wordinterval-sort w  $\equiv$  l2wi (mergesort-remdups (wi2l w))

```

```

lemma wordinterval-sort: wordinterval-to-set (wordinterval-sort w) = wordinterval-to-set w
by (simp add: wordinterval-sort-def wi2l l2wi mergesort-remdups-correct)

```

```

end
theory IP-Address-Parser
imports IP-Address
         IPv4
         IPv6
         HOL-Library.Code-Target-Nat
begin

```

9 Parsing IP Addresses

9.1 IPv4 Parser

```

ML
local
  fun extract-int ss = case ss |> implode |> Int.fromString
    of SOME i => i
    | NONE => raise Fail unparsable int;

  fun mk-nat maxval i = if i < 0 orelse i > maxval
    then
      raise Fail(nat ( $\wedge$ Int.toString i $\wedge$ ) must be between 0 and  $\wedge$ Int.toString
maxval)

```

```

      else (HOLogic.mk-number HOLogic.natT i);
    val mk-nat255 = mk-nat 255;

    fun mk-quadrupel (((a,b),c),d) = HOLogic.mk-prod
      (mk-nat255 a, HOLogic.mk-prod (mk-nat255 b, HOLogic.mk-prod (mk-nat255
c, mk-nat255 d)));

  in
    fun mk-ipv4addr ip = @{const ipv4addr-of-dotdecimal} $ mk-quadrupel ip;

    val parser-ipv4 = (Scan.many1 Symbol.is-ascii-digit >> extract-int) --| (($ .)
--
      (Scan.many1 Symbol.is-ascii-digit >> extract-int) --| (($ .) --
      (Scan.many1 Symbol.is-ascii-digit >> extract-int) --| (($ .) --
      (Scan.many1 Symbol.is-ascii-digit >> extract-int));

  end;

local
  val (ip-term, rest) = 10.8.0.255 |> raw-explode |> Scan.finite Symbol.stopper
(parser-ipv4 >> mk-ipv4addr);
in
  val - = if rest <> [] then raise Fail did not parse everything else writeln parsed;
  val - = if
    Code-Evaluation.dynamic-value-strict @ {context} ip-term
    <> @ {term 168296703::ipv4addr}
  then
    raise Fail parser failed
  else
    writeln test passed;
end;
>

```

9.2 IPv6 Parser

definition *mk-ipv6addr* :: 16 word option list \Rightarrow *ipv6addr-syntax* option **where**
mk-ipv6addr partslist = (
 let — remove empty lists to the beginning and end if omission occurs at
 start/end
 — to join over : properly
 fix-start = (λps . case ps of None#None#- \Rightarrow tl ps | - \Rightarrow ps);
 fix-end = (λps . case rev ps of None#None#- \Rightarrow butlast ps | - \Rightarrow ps);
 ps = (fix-end \circ fix-start) partslist
in
if length (filter (λp . p = None) ps) = 1
then *ipv6-unparsed-compressed-to-preferred* ps
else case ps of [Some a,Some b,Some c,Some d,Some e,Some f,Some g,Some
h]
 \Rightarrow Some (IPv6AddrPreferred a b c d e f g h)
| - \Rightarrow None

```

)

ML<
local
  val fromHexString = StringCvt.scanString (Int.scan StringCvt.HEX);

  fun extract-int ss = case ss of => NONE
    | xs =>
      case xs |> fromHexString
        of SOME i => SOME i
         | NONE   => raise Fail unparsable int;

in
  val mk-ipv6addr = map (fn p => case p of NONE => @ {const None (16 word)}
    | SOME i => @ {const Some (16 word)} $
      (@ {const of-int (16 word)} $
        HOLogic.mk-number
          HOLogic.intT i)
    )
    #> HOLogic.mk-list @ {typ 16 word option}
    (* TODO: never use THE! is there some option-dest?*)
    #> (fn x => @ {const ipv6preferred-to-int} $
      (@ {const the (ipv6addr-syntax)} $ (@ {const mk-ipv6addr}
        $ x)));

  val parser-ipv6 = Scan.many1 (fn x => Symbol.is-ascii-hex x orelse x = :)
    >> (implode #> space-explode : #> map extract-int)
    (* a different implementation which returns a list of exploded strings:
      Scan.repeat ((Scan.many Symbol.is-ascii-hex >> extract-int) --|
        ($$ :))
        @@@ (Scan.many Symbol.is-ascii-hex >> extract-int >> (fn p =>
          [p]))*)
  end;

local
  val parse-ipv6 = raw-explode
    #> Scan.finite Symbol.stopper (parser-ipv6 >> mk-ipv6addr);
  fun unit-test (ip-string, ip-result) = let
    val (ip-term, rest) = ip-string |> parse-ipv6;
    val - = if rest <> [] then raise Fail did not parse everything else ();
    val - = Code-Evaluation.dynamic-value-strict @ {context} ip-term |> Syn-
      tax.pretty-term @ {context} |> Pretty.writeln;
    val - = if
      Code-Evaluation.dynamic-value-strict @ {context} ip-term <> ip-result
    then
      raise Fail parser failed
    else
      writeln (test passed for ^ip-string);
in

```

```

    ()
  end;
in
  val - = map unit-test
    [(10:ab:FF:0::FF:4:255, @ {term 83090298060623265259947972050027093::ipv6addr})
    ,(2001:db8::8:800:200c:417a, @ {term 42540766411282592856906245548098208122::ipv6addr})
    ,(ff01::101, @ {term 338958331222012082418099330867817087233::ipv6addr})
    ,(::8:800:200c:417a, @ {term 2260596444381562::ipv6addr})
    ,(2001:db8::, @ {term 42540766411282592856903984951653826560::ipv6addr})
    ,(ff00::, @ {term 338953138925153547590470800371487866880::ipv6addr})
    ,(fe80::, @ {term 338288524927261089654018896841347694592::ipv6addr})
    ,(1::, @ {term 5192296858534827628530496329220096::ipv6addr})
    ,(1::, @ {term 5192296858534827628530496329220096::ipv6addr})
    ,(::, @ {term 0::ipv6addr})
    ,(::1, @ {term 1::ipv6addr})
    ,(2001:db8:0:1:1:1:1, @ {term 42540766411282592875351010504635121665::ipv6addr})
    ,(fff:fff:fff:fff:fff:fff:fff:fff, @ {term 340282366920938463463374607431768211455::ipv6addr})
    ];
  end;
>
end
theory Lib-Numbers-toString
imports Main
begin

```

10 Printing Numbers

```

fun string-of-nat :: nat ⇒ string where
  string-of-nat n = (if n < 10 then [char-of (48 + n)] else
    string-of-nat (n div 10) @ [char-of (48 + (n mod 10))])
definition string-of-int :: int ⇒ string where
  string-of-int i = (if i < 0 then "-" @ string-of-nat (nat (- i)) else
    string-of-nat (nat i))

```

lemma string-of-nat 123456 = "123456" **by** eval

declare string-of-nat.simps[simp del]

```

end
theory Lib-Word-toString
imports Lib-Numbers-toString
  Word-Lib.Word-Lemmas
begin

```

context linordered-euclidean-semiring
begin

```

lemma exp-estimate [simp]:
  ⟨numeral Num.One ≤ 2 ^ n⟩ (is ⟨?P1⟩)

```

```

  ⟨numeral Num.One < 2 ^ n ↔ 1 ≤ n⟩ (is ⟨?P2⟩)
  ⟨numeral (Num.Bit0 k) ≤ 2 ^ n ↔ 1 ≤ n ∧ numeral k ≤ 2 ^ (n - 1)⟩ (is
  ⟨?P3⟩)
  ⟨numeral (Num.Bit0 k) < 2 ^ n ↔ 1 ≤ n ∧ numeral k < 2 ^ (n - 1)⟩ (is
  ⟨?P4⟩)
  ⟨numeral (Num.Bit1 k) ≤ 2 ^ n ↔ 1 ≤ n ∧ numeral k < 2 ^ (n - 1)⟩ (is
  ⟨?P5⟩)
  ⟨numeral (Num.Bit1 k) < 2 ^ n ↔ 1 ≤ n ∧ numeral k < 2 ^ (n - 1)⟩ (is
  ⟨?P6⟩)
proof -
  show ?P1
    by simp
  show ?P2
    using one-less-power power-eq-if by auto

  let ?K = ⟨numeral k :: nat⟩

  define m where ⟨m = n - 1⟩
  then consider ⟨n = 0⟩ | ⟨n = Suc m⟩
    by (cases n) simp-all
  note Suc = this

  have ⟨2 * ?K ≤ 2 * 2 ^ m ↔ ?K ≤ 2 ^ m⟩
    by linarith
  then have ⟨of-nat (2 * ?K) ≤ of-nat (2 * 2 ^ m) ↔ of-nat ?K ≤ of-nat (2 ^
  m)⟩
    by (simp only: of-nat-le-iff)
  then show ?P3
    by (auto intro: Suc)

  have ⟨2 * ?K < 2 * 2 ^ m ↔ ?K < 2 ^ m⟩
    by linarith
  then have ⟨of-nat (2 * ?K) < of-nat (2 * 2 ^ m) ↔ of-nat ?K < of-nat (2 ^
  m)⟩
    by (simp only: of-nat-less-iff)
  then show ?P4
    by (auto intro: Suc)

  have ⟨Suc (2 * ?K) ≤ 2 * 2 ^ m ↔ ?K < 2 ^ m⟩
    by linarith
  then have ⟨of-nat (Suc (2 * ?K)) ≤ of-nat (2 * 2 ^ m) ↔ of-nat ?K < of-nat
  (2 ^ m)⟩
    by (simp only: of-nat-le-iff of-nat-less-iff)
  then show ?P5
    by (auto intro: Suc)

  have ⟨Suc (2 * ?K) < 2 * 2 ^ m ↔ ?K < 2 ^ m⟩
    by linarith
  then have ⟨of-nat (Suc (2 * ?K)) < of-nat (2 * 2 ^ m) ↔ of-nat ?K < of-nat

```

```

(2 ^ m)
  by (simp only: of-nat-less-iff)
  then show ?P6
  by (auto intro: Suc)

```

qed

end

11 Printing Machine Words

definition *string-of-word-single* :: *bool* \Rightarrow *'a::len word* \Rightarrow *string* **where**
string-of-word-single *lc w* \equiv
 (if
 w < 10
 then
 [*char-of* (48 + *unat w*)]
 else if
 w < 36
 then
 [*char-of* ((if *lc* then 87 else 55) + *unat w*)]
 else
 undefined)

Example:

lemma *let word-upto* = ((λ *i j*. *map* (*of-nat* \circ *nat*) [*i .. j*])) :: *int* \Rightarrow *int* \Rightarrow 32 *word list*
in map (*string-of-word-single* *False*) (*word-upto* 1 35) =
 ["1", "2", "3", "4", "5", "6", "7", "8", "9",
 "A", "B", "C", "D", "E", "F", "G", "H", "I",
 "J", "K", "L", "M", "N", "O", "P", "Q", "R",
 "S", "T", "U", "V", "W", "X", "Y", "Z"] **by** *eval*

function *string-of-word* :: *bool* \Rightarrow (*'a :: len*) *word* \Rightarrow *nat* \Rightarrow (*'a :: len*) *word* \Rightarrow *string* **where**
string-of-word *lc base ml w* =
 (if
 base < 2 \vee *LENGTH*('a) < 2
 then
 undefined
 else if
 w < *base* \wedge *ml* = 0
 then
 string-of-word-single *lc w*
 else
 string-of-word *lc base* (*ml* - 1) (*w div base*) @ *string-of-word-single* *lc* (*w mod base*)
)

by *pat-completeness auto*

definition *hex-string-of-word* $l \equiv \text{string-of-word True } (16 :: ('a::\text{len}) \text{ word}) \ l$

definition *hex-string-of-word0* $\equiv \text{hex-string-of-word } 0$

definition *dec-string-of-word0* $\equiv \text{string-of-word True } 10 \ 0$

termination *string-of-word*

apply(*relation measure* $(\lambda(a,b,c,d). \text{unat } d + c)$)

apply(*rule wf-measure*)

apply(*subst in-measure*)

apply(*clarsimp*)

subgoal for *base ml n*

apply(*case-tac ml $\neq 0$*)

apply(*simp add: less-eq-Suc-le unat-div*)

apply(*simp*)

apply(*subgoal-tac (n div base) < n*)

apply(*blast intro: unat-mono*)

apply(*rule div-less-dividend-word*)

apply (*auto simp add: not-less word-le-nat-alt*)

done

done

declare *string-of-word.simps*[*simp del*]

lemma *hex-string-of-word0* (*0xdeadbeef42* :: *42 word*) = "*deadbeef42*" **by** *eval*

lemma *hex-string-of-word 1* (*0x1* :: *5 word*) = "*01*" **by** *eval*

lemma *hex-string-of-word 8* (*0xff*::*32 word*) = "*0000000ff*" **by** *eval*

lemma *dec-string-of-word0* (*8*::*32 word*) = "*8*" **by** *eval*

lemma *dec-string-of-word0* (*3*::*2 word*) = "*11*" **by** *eval*

lemma *dec-string-of-word0* (*-1*::*8 word*) = "*255*" **by** *eval*

lemma *string-of-word-single-atoi*:

$n < 10 \implies \text{string-of-word-single True } n = [\text{char-of } (48 + \text{unat } n)]$

by(*simp add: string-of-word-single-def*)

lemma *bintrunc-pos-eq*: $x \geq 0 \implies \text{take-bit } n \ x = x \iff x < 2^{\wedge}n$ **for** $x :: \text{int}$

by (*simp add: take-bit-int-eq-self-iff*)

lemma *string-of-word-base-ten-zeropad*:

fixes $w :: 'a::\text{len} \ \text{word}$

assumes *lena*: $\text{LENGTH}('a) \geq 5$

shows $\text{base} = 10 \implies \text{zero} = 0 \implies \text{string-of-word True base zero } w = \text{string-of-nat } (\text{unat } w)$

proof(*induction True base zero w rule: string-of-word.induct*)

```

case (1 base ml n)

note Word.word-less-no[simp del]
note Word.uint-bintrunc[simp del]

define l where ⟨l = LENGTH('a) - 5⟩
with lena have l: ⟨LENGTH('a) = l + 5⟩
  by simp

have [simp]: ⟨take-bit LENGTH('a) (10 :: nat) = 10⟩
  using lena by (auto simp add: take-bit-nat-eq-self-iff l Suc-lessI)

have [simp]: ⟨take-bit LENGTH('a) (10 :: int) = 10⟩
  using lena by (auto simp add: take-bit-int-eq-self-iff l)
  (smt (verit) zero-less-power)

have unat-mod-ten: unat (n mod 0xA) = unat n mod 10
  by (simp add: nat-take-bit-eq unat-mod)

have unat-div-ten: (unat (n div 0xA)) = unat n div 10
  by (simp add: nat-take-bit-eq unat-div)

have n-less-ten-unat: n < 0xA ⟹ (unat n < 10)
  by (simp add: unat-less-helper)

have 0xA ≤ n ⟹ 10 ≤ unat n
  by (simp add: nat-take-bit-eq word-le-nat-alt)

hence n-less-ten-unat-not: ¬ n < 0xA ⟹ ¬ unat n < 10 by fastforce
have not-wordlength-too-small: ¬ LENGTH('a) < 2 using lena by fastforce
have 2 ≤ (0xA::'a word)
  by simp
hence ten-not-less-two: ¬ (0xA::'a word) < 2 by (simp add: Word.word-less-no
Word.uint-bintrunc)
with 1(2,3) have ¬ (base < 2 ∨ LENGTH(32) < 2)
  by(simp)
with 1 not-wordlength-too-small have IH: ¬ n < 0xA ⟹ string-of-word True
0xA 0 (n div 0xA) = string-of-nat (unat (n div 0xA))
  by(simp)
show ?case
  apply(simp add: 1)
  apply(cases n < 0xA)
  subgoal
    apply(subst(1) string-of-word.simps)
    apply(subst(1) string-of-nat.simps)
    apply(simp add: n-less-ten-unat)
    using lena apply(simp add: not-wordlength-too-small ten-not-less-two string-of-word-single-atoi)
  done
using sym[OF IH] apply(simp)

```

```

    apply(subst(1) string-of-word.simps)
    apply(simp)
    apply(subst(1) string-of-nat.simps)
    apply(simp)
    apply (simp add: string-of-word-single-atoi Word.word-mod-less-divisor unat-div-ten
    unat-mod-ten)
    using <math>10 \leq n \implies 10 \leq \text{unat } n</math> not-wordlength-too-small apply (auto simp
    add: not-less)
    done
  qed

```

```

lemma dec-string-of-word0:
  dec-string-of-word0 (w8:: 8 word) = string-of-nat (unat w8)
  dec-string-of-word0 (w16:: 16 word) = string-of-nat (unat w16)
  dec-string-of-word0 (w32:: 32 word) = string-of-nat (unat w32)
  dec-string-of-word0 (w64:: 64 word) = string-of-nat (unat w64)
  dec-string-of-word0 (w128:: 128 word) = string-of-nat (unat w128)
  unfolding dec-string-of-word0-def
  using string-of-word-base-ten-zeropad by force+

```

```

end
theory Lib-List-toString
imports Lib-Numbers-toString
begin

```

12 Printing Lists

```

fun intersperse :: 'a  $\Rightarrow$  'a list list  $\Rightarrow$  'a list where
  intersperse - [] = [] |
  intersperse a [x] = x |
  intersperse a (x#xs) = x @ a # intersperse a xs

```

```

definition list-separated-toString :: string  $\Rightarrow$  ('a  $\Rightarrow$  string)  $\Rightarrow$  'a list  $\Rightarrow$  string
where
  list-separated-toString sep toStr ls = concat (splice (map toStr ls) (replicate (length
  ls - 1) sep))

```

A slightly more efficient code equation, which is actually not really faster (in certain languages)

```

fun list-separated-toString-helper :: string  $\Rightarrow$  ('a  $\Rightarrow$  string)  $\Rightarrow$  'a list  $\Rightarrow$  string
where
  list-separated-toString-helper sep toStr [] = "" |
  list-separated-toString-helper sep toStr [l] = toStr l |
  list-separated-toString-helper sep toStr (l#ls) = (toStr l)@sep@list-separated-toString-helper
  sep toStr ls
lemma list-separated-toString-helper: list-separated-toString = list-separated-toString-helper
proof -

```

```

{ fix sep and toStr::('a ⇒ char list) and ls
  have list-separated-toString sep toStr ls = list-separated-toString-helper sep toStr
ls
  by(induction sep toStr ls rule: list-separated-toString-helper.induct) (simp-all
add: list-separated-toString-def)
} thus ?thesis by(simp add: fun-eq-iff)
qed

```

```

lemma list-separated-toString-intersperse:
intersperse sep (map f xs) = list-separated-toString [sep] f xs
apply(simp add: list-separated-toString-helper)
apply(induction [sep] f xs rule: list-separated-toString-helper.induct)
by simp+

```

```

definition list-toString :: ('a ⇒ string) ⇒ 'a list ⇒ string where
list-toString toStr ls = "[@" list-separated-toString ", " toStr ls "@" "]"

```

```

lemma list-toString string-of-nat [1,2,3] = "[1, 2, 3]" by eval

```

```

end
theory IP-Address-toString
imports IP-Address IPv4 IPv6
  Lib-Word-toString
  Lib-List-toString
  HOL-Library.Code-Target-Nat
begin

```

13 Pretty Printing IP Addresses

13.1 Generic Pretty Printer

Generic function. Whenever possible, use IPv4 or IPv6 pretty printing!

```

definition ipaddr-generic-toString :: 'i::len word ⇒ string where
ipaddr-generic-toString ip ≡
"[IP address (" @ string-of-nat (LENGTH('i)) @ " bit): " @ dec-string-of-word0
ip @ "]"

```

```

lemma ipaddr-generic-toString (ipv4addr-of-dotdecimal (192,168,0,1)) = "[IP
address (32 bit): 3232235521]" by eval

```

13.2 IPv4 Pretty Printing

```

fun dotteddecimal-toString :: nat × nat × nat × nat ⇒ string where
dotteddecimal-toString (a,b,c,d) =
string-of-nat a@"."@string-of-nat b@"."@string-of-nat c@"."@string-of-nat d

```

```

definition ipv4addr-toString :: ipv4addr ⇒ string where
ipv4addr-toString ip = dotteddecimal-toString (dotdecimal-of-ipv4addr ip)

```

lemma *ipv4addr-toString (ipv4addr-of-dotdecimal (192, 168, 0, 1)) = "192.168.0.1"*
by *eval*

Correctness Theorems:

thm *dotdecimal-of-ipv4addr-ipv4addr-of-dotdecimal
 ipv4addr-of-dotdecimal-dotdecimal-of-ipv4addr*

13.3 IPv6 Pretty Printing

definition *ipv6addr-toString :: ipv6addr ⇒ string where*
ipv6addr-toString ip = (
let partslst = ipv6-preferred-to-compressed (int-to-ipv6preferred ip);
— add empty lists to the beginning and end if omission occurs at start/end
— to join over : properly
fix-start = (λps. case ps of None#- ⇒ None#ps | - ⇒ ps);
fix-end = (λps. case rev ps of None#- ⇒ ps@[None] | - ⇒ ps)
in list-separated-toString "."
(λpt. case pt of None ⇒ ""
| Some w ⇒ hex-string-of-word0 w)
((fix-end ∘ fix-start) partslst)
)

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0xDB8
 0x0 0x0 0x8 0x800 0x200C 0x417A))*
= "2001:db8::8:800:200c:417a" **by** *eval* — a unicast address

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0xFF01 0x0
 0x0 0x0 0x0 0x0 0x0 0x0101)) =*
"ff01::101" **by** *eval* — a multicast address

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0 0 0 0 0x8
 0x800 0x200C 0x417A)) =*
"::8:800:200c:417a" **by** *eval*

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0xDB8
 0 0 0 0 0)) =*
"2001:db8::" **by** *eval*

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0xFF00 0 0
 0 0 0 0)) =*
"ff00::" **by** *eval* — Multicast

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0xFE80 0 0
 0 0 0 0)) =*
"fe80::" **by** *eval* — Link-Local unicast

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0 0 0 0 0 0 0
 0)) =*

":::1" by eval — unspecified address

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0 0 0 0 0 0 1)) =*

":::1" by eval — loopback address

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0xdb8 0x0 0x0 0x0 0x0 0x1)) =*
 "2001:db8::1" by eval — Section 4.1 of RFC5952

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0xdb8 0x0 0x0 0x0 0x0 0x2 0x1)) =*
 "2001:db8::2:1" by eval — Section 4.2.1 of RFC5952

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0xdb8 0x0 0x1 0x1 0x1 0x1 0x1)) =*
 "2001:db8:0:1:1:1:1:1" by eval — Section 4.2.2 of RFC5952

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0x0 0x0 0x1 0x0 0x0 0x0 0x1)) =*
 "2001:0:0:1::1" by eval — Section 4.2.3 of RFC5952

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0xdb8 0x0 0x0 0x1 0x0 0x0 0x1)) =*
 "2001:db8::1:0:0:1" by eval — Section 4.2.3 of RFC5952

lemma *ipv6addr-toString max-ipv6-addr = "ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff" by eval*

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0xffff 0xffff 0xffff 0xffff 0xffff 0xffff 0xffff 0xffff)) =*
 "ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff" by eval

Correctness Theorems:

thm *ipv6-preferred-to-compressed*
ipv6-preferred-to-compressed-RFC-4291-format
ipv6-unparsed-compressed-to-preferred-identity1
ipv6-unparsed-compressed-to-preferred-identity2
RFC-4291-format
ipv6preferred-to-int-int-to-ipv6preferred
int-to-ipv6preferred-ipv6preferred-to-int

end

theory *Prefix-Match-toString*

imports *IP-Address-toString Prefix-Match*

begin

definition *prefix-match-32-toString :: 32 prefix-match ⇒ string where*
prefix-match-32-toString pfx = (case pfx of PrefixMatch p l ⇒ ipv4addr-toString

p @ (if $l \neq 32$ then $"/"$ @ string-of-nat l else [])
definition *prefix-match-128-toString* :: 128 *prefix-match* \Rightarrow string **where**
 prefix-match-128-toString px = (case px of *PrefixMatch* p l \Rightarrow *ipv6addr-toString*
 p @ (if $l \neq 128$ then $"/"$ @ string-of-nat l else []))

end