

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	7
2	Definitions inspired by the Haskell World.	13
3	Modelling IP Addresses	14
3.1	Sets of IP Addresses	14
3.2	IP Addresses as WordIntervals	16
3.3	IP Addresses in CIDR Notation	17
3.4	Clever Operations on IP Addresses in CIDR Notation	18
3.5	Code Equations	20

4	IPv4 Addresses	21
4.1	Representing IPv4 Addresses (Syntax)	21
4.2	IP Ranges: Examples	22
5	IPv6 Addresses	24
5.1	Syntax of IPv6 Addresses	25
5.2	Semantics	33
5.3	IPv6 Pretty Printing (converting to compressed format) . . .	37
6	Prefix Match	41
6.1	Address Semantics	42
6.2	Relation between prefix and set	43
6.3	Equivalence Proofs	43
7	CIDR Split Motivation (Example for IPv4)	44
8	CIDR Split	45
8.1	Versions for <i>ipset-from-cidr</i>	47
9	Parsing IP Addresses	49
9.1	IPv4 Parser	49
9.2	IPv6 Parser	49
10	Printing Numbers	49
11	Printing Machine Words	50
12	Printing Lists	52
13	Pretty Printing IP Addresses	53
13.1	Generic Pretty Printer	53
13.2	IPv4 Pretty Printing	53
13.3	IPv6 Pretty Printing	53

```

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)
  <proof>
```

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  $\cup$  wordinter-
val-to-set r2
  <proof>
```

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

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

wordinterval-empty (*RangeUnion* *r1 r2*) \longleftrightarrow *wordinterval-empty* *r1* \wedge *wordinterval-empty* *r2*

lemma *wordinterval-empty-set-eq[simp]*: *wordinterval-empty* *r* \longleftrightarrow *wordinterval-to-set* *r* = $\{\}$
 \langle *proof* \rangle

definition *Empty-WordInterval* :: 'a::len *wordinterval* **where**

Empty-WordInterval \equiv *WordInterval* 1 0

lemma *wordinterval-empty-Empty-WordInterval*: *wordinterval-empty* *Empty-WordInterval*
 \langle *proof* \rangle

lemma *Empty-WordInterval-set-eq[simp]*: *wordinterval-to-set* *Empty-WordInterval*
 = $\{\}$
 \langle *proof* \rangle

1.4 WordInterval and Lists

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

wordinterval to list

fun *wi2l* :: 'a::len *wordinterval* \Rightarrow ('a::len *word* \times 'a::len *word*) *list* **where**
wi2l (*RangeUnion* *r1 r2*) = *wi2l* *r1* @ *wi2l* *r2* |
wi2l (*WordInterval* *s e*) = (if *e* < *s* then [] else [(*s*,*e*)])

list to *wordinterval*

fun *l2wi* :: ('a::len *word* \times 'a *word*) *list* \Rightarrow 'a *wordinterval* **where**
l2wi [] = *Empty-WordInterval* |
l2wi [(*s*,*e*)] = (*WordInterval* *s e*) |
l2wi ((*s*,*e*)#*rs*) = (*RangeUnion* (*WordInterval* *s e*) (*l2wi* *rs*))

lemma *l2wi-append*: *wordinterval-to-set* (*l2wi* (*l1*@*l2*)) =
wordinterval-to-set (*l2wi* *l1*) \cup *wordinterval-to-set* (*l2wi* *l2*)
 \langle *proof* \rangle

lemma *l2wi-wi2l[simp]*: *wordinterval-to-set* (*l2wi* (*wi2l* *r*)) = *wordinterval-to-set*
r
 \langle *proof* \rangle

lemma *l2wi*: *wordinterval-to-set* (*l2wi* *l*) = (\bigcup (*i*,*j*) \in *set* *l*. {*i* .. *j*})
 \langle *proof* \rangle

lemma *wi2l*: (\bigcup (*i*,*j*) \in *set* (*wi2l* *r*). {*i* .. *j*}) = *wordinterval-to-set* *r*
 \langle *proof* \rangle

lemma *l2wi-remdups[simp]*: *wordinterval-to-set* (*l2wi* (*remdups* *ls*)) = *wordinterval-to-set* (*l2wi* *ls*)
 \langle *proof* \rangle

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


```

wordinterval-optimize-empty2 r = r
lemma wordinterval-optimize-empty-code[code-unfold]:
wordinterval-optimize-empty = wordinterval-optimize-empty2
⟨proof⟩
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')
⟨proof⟩ lemma [code]:
disjoint-intervals A B =
(case A of (s,e) ⇒ case B of (s',e') ⇒ s > e' ∨ s' > e ∨ s > e ∨ s' > e')
⟨proof⟩

```

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'}

```

⟨proof⟩ **lemma** *disjoint-subset*: $\text{disjoint } A B \implies A \subseteq B \cup C \implies A \subseteq C$
 ⟨proof⟩ **lemma** *merge-overlap-helper1*: $\text{interval-of } A \subseteq (\bigcup s \in \text{set } ss. \text{interval-of } s) \implies$
 $(\bigcup s \in \text{set } (\text{merge-overlap } A \text{ } ss). \text{interval-of } s) = (\bigcup s \in \text{set } ss. \text{interval-of } s)$
 ⟨proof⟩ **lemma** *merge-overlap-helper2*: $\exists s' \in \text{set } ss. \neg \text{disjoint } (\text{interval-of } A)$
 $(\text{interval-of } s') \implies$
 $\text{interval-of } A \cup (\bigcup s \in \text{set } ss. \text{interval-of } s) = (\bigcup s \in \text{set } (\text{merge-overlap } A$
 $ss). \text{interval-of } s)$
 ⟨proof⟩ **lemma** *merge-overlap-length*:
 $\exists s' \in \text{set } ss. \neg \text{disjoint } (\text{interval-of } A) (\text{interval-of } s') \implies$
 $\text{length } (\text{merge-overlap } A \text{ } ss) = \text{length } ss$
 ⟨proof⟩

lemma *merge-overlap* (1::16 word,2) [(1, 7)] = [(1, 7)] ⟨proof⟩

lemma *merge-overlap* (1::16 word,2) [(2, 7)] = [(1, 7)] ⟨proof⟩

lemma *merge-overlap* (1::16 word,2) [(3, 7)] = [(3, 7), (1,2)] ⟨proof⟩ **function**
listwordinterval-compress

:: (('a::len) word × ('a::len) word) list ⇒ ('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))

⟨proof⟩ **termination** *listwordinterval-compress*

⟨proof⟩ **lemma** *listwordinterval-compress*:

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

⟨proof⟩

lemma *listwordinterval-compress* [(1::32 word,3), (8,10), (2,5), (3,7)] = [(8,10), (1, 7)]

⟨proof⟩ **lemma** *A-in-listwordinterval-compress*: $A \in \text{set } (\text{listwordinterval-compress } ss) \implies$

$\text{interval-of } A \subseteq (\bigcup s \in \text{set } ss. \text{interval-of } s)$

⟨proof⟩ **lemma** *listwordinterval-compress-disjoint*:

$A \in \text{set } (\text{listwordinterval-compress } ss) \implies B \in \text{set } (\text{listwordinterval-compress } ss) \implies$

$A \neq B \implies \text{disjoint } (\text{interval-of } A) (\text{interval-of } B)$

⟨proof⟩

END merging overlapping intervals

BEGIN merging adjacent intervals

private fun *merge-adjacent*

:: (('a::len) word × ('a::len) word) ⇒ ('a word × 'a word) list ⇒ ('a word × 'a word) 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*:

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

<proof> **lemma** *merge-adjacent-length*:

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

<proof> **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 $[] = [] \mid$

listwordinterval-adjacent $((s, e)\#ss) = ($

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

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

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

<proof> **termination** *listwordinterval-adjacent*

<proof> **lemma** *listwordinterval-adjacent*:

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

<proof>

lemma *listwordinterval-adjacent* $[(1::16 \text{word}, 3), (5, 10), (10, 10), (4, 4)] = [(10, 10), (1, 10)]$

<proof>

END merging adjacent intervals

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

wordinterval-compress $r \equiv$

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

Correctness: Compression preserves semantics

lemma *wordinterval-compress*:

$\text{wordinterval-to-set } (\text{wordinterval-compress } r) = \text{wordinterval-to-set } r$

<proof>

end

Example

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

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

$=$

$[(56, 80001), (0, 10)]$ *<proof>*

lemma *wordinterval-compress* (*RangeUnion* (*RangeUnion* (*WordInterval* (1::32
word) 5)
(*WordInterval* 8 10)) (*WordInterval* 3
7)) =
WordInterval 1 10 *<proof>*

1.6 Further operations

U

definition *wordinterval-Union* :: ('a::len) *wordinterval list* ⇒ 'a *wordinterval*
where
wordinterval-Union ws = *wordinterval-compress* (*foldr wordinterval-union ws*
Empty-WordInterval)

lemma *wordinterval-Union*:
wordinterval-to-set (*wordinterval-Union ws*) = (U *w* ∈ (*set ws*). *wordinter-*
val-to-set w)
<proof>

context

begin

private fun *wordinterval-setminus'*

:: 'a::len *wordinterval* ⇒ 'a *wordinterval* ⇒ 'a *wordinterval* **where**
wordinterval-setminus' (*WordInterval s e*) (*WordInterval ms me*) = (
if *s > e* ∨ *ms > me* then *WordInterval s e* else
if *me* ≥ *e*
then
WordInterval (if *ms* = 0 then 1 else *s*) (*min e* (*word-prev ms*))
else if *ms* ≤ *s*
then
WordInterval (*max s* (*word-next me*)) (if *me* = - 1 then 0 else *e*)
else
RangeUnion (*WordInterval* (if *ms* = 0 then 1 else *s*) (*word-prev ms*))
(*WordInterval* (*word-next me*) (if *me* = - 1 then 0 else *e*))
) |
wordinterval-setminus' (*RangeUnion r1 r2*) *t* =
RangeUnion (*wordinterval-setminus'* *r1 t*) (*wordinterval-setminus'* *r2 t*) |
wordinterval-setminus' *t* (*RangeUnion r1 r2*) =
wordinterval-setminus' (*wordinterval-setminus'* *t r1*) *r2*

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

wordinterval-to-set(*wordinterval-setminus'* (*WordInterval s e*) (*WordInterval ms*
me)) =
wordinterval-to-set (*WordInterval s e*) - *wordinterval-to-set* (*WordInterval ms*
me)
<proof> **lemma** *wordinterval-setminus'-set-eq*:
wordinterval-to-set (*wordinterval-setminus'* *r1 r2*) =

```

    wordinterval-to-set r1 - wordinterval-to-set r2
  <proof>
lemma wordinterval-setminus'-empty-struct:
  wordinterval-empty r2  $\implies$  wordinterval-setminus' r1 r2 = r1
  <proof>

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
  <proof>
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
  <proof>

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 <proof>

lemma wordinterval-invert-UNIV-empty:
  wordinterval-empty (wordinterval-invert wordinterval-UNIV) <proof>

lemma wi2l-univ[simp]: wi2l wordinterval-UNIV = [(0, - 1)]
  <proof>

 $\cap$ 

context
begin
  private lemma  $\{(s::nat) .. e\} \cap \{s' .. e'\} = \{\}$   $\longleftrightarrow$   $s > e' \vee s' > e \vee s > e \vee$ 
 $s' > e'$ 
  <proof> 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')
  ) |

```

$\text{wordinterval-intersection}' (\text{RangeUnion } r1 \ r2) \ t =$
 $\text{RangeUnion } (\text{wordinterval-intersection}' \ r1 \ t) \ (\text{wordinterval-intersection}' \ r2$
 $t)$
 $\text{wordinterval-intersection}' \ t \ (\text{RangeUnion } r1 \ r2) =$
 $\text{RangeUnion } (\text{wordinterval-intersection}' \ t \ r1) \ (\text{wordinterval-intersection}' \ t$
 $r2)$

private lemma *wordinterval-intersection'-set-eq*:
 $\text{wordinterval-to-set } (\text{wordinterval-intersection}' \ r1 \ r2) =$
 $\text{wordinterval-to-set } r1 \ \cap \ \text{wordinterval-to-set } r2$
 $\langle \text{proof} \rangle$

lemma *wordinterval-intersection'*
 $(\text{RangeUnion } (\text{RangeUnion } (\text{WordInterval } (1::32 \ \text{word}) \ 3) \ (\text{WordInterval}$
 $8 \ 10)))$
 $(\text{WordInterval } 1 \ 3)) \ (\text{WordInterval } 1 \ 3) =$
 $\text{RangeUnion } (\text{RangeUnion } (\text{WordInterval } 1 \ 3) \ (\text{WordInterval } 1 \ 0))$
 $(\text{WordInterval } 1 \ 3) \ \langle \text{proof} \rangle$

definition *wordinterval-intersection*
 $:: 'a::\text{len } \text{wordinterval} \Rightarrow 'a::\text{len } \text{wordinterval} \Rightarrow 'a::\text{len } \text{wordinterval} \ \mathbf{where}$
 $\text{wordinterval-intersection } r1 \ r2 \equiv \text{wordinterval-compress } (\text{wordinterval-intersection}'$
 $r1 \ r2)$

lemma *wordinterval-intersection-set-eq[simp]*:
 $\text{wordinterval-to-set } (\text{wordinterval-intersection } r1 \ r2) =$
 $\text{wordinterval-to-set } r1 \ \cap \ \text{wordinterval-to-set } r2$
 $\langle \text{proof} \rangle$

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

end

definition *wordinterval-subset* $:: 'a::\text{len } \text{wordinterval} \Rightarrow 'a::\text{len } \text{wordinterval} \Rightarrow \text{bool}$
where

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

lemma *wordinterval-subset-set-eq[simp]*:

$\text{wordinterval-subset } r1 \ r2 = (\text{wordinterval-to-set } r1 \ \subseteq \ \text{wordinterval-to-set } r2)$
 $\langle \text{proof} \rangle$

definition *wordinterval-eq* $:: 'a::\text{len } \text{wordinterval} \Rightarrow 'a::\text{len } \text{wordinterval} \Rightarrow \text{bool}$
where

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

lemma *wordinterval-eq-set-eq*:

$\text{wordinterval-eq } r1 \ r2 \ \longleftrightarrow \ \text{wordinterval-to-set } r1 = \text{wordinterval-to-set } r2$

<proof>

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

lemma *wordinterval-eq-comm*: *wordinterval-eq r1 r2* \longleftrightarrow *wordinterval-eq r2 r1*

<proof>

lemma *wordinterval-to-set-alt*: *wordinterval-to-set r* = {*x. wordinterval-element x r*}

<proof>

lemma *wordinterval-un-empty*:

wordinterval-empty r1 \implies *wordinterval-eq (wordinterval-union r1 r2) r2*

<proof>

lemma *wordinterval-un-empt-b*:

wordinterval-empty r2 \implies *wordinterval-eq (wordinterval-union r1 r2) r1*

<proof>

lemma *wordinterval-Diff-triv*:

wordinterval-empty (wordinterval-intersection a b) \implies *wordinterval-eq (wordinterval-setminus a b) a*

<proof>

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)*

<proof>

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

<proof>

lemma *wordinterval-eq-reflp*:

reflp wordinterval-eq

<proof>

lemma *wordinterval-eq-symp*:

symp wordinterval-eq

<proof>

lemma *wordinterval-eq-transp*:

transp wordinterval-eq

<proof>

lemma *wordinterval-eq-equivp*:

equivp wordinterval-eq

<proof>

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 = Inf S \Longrightarrow *is-lowest-element* x S

<proof>

lemma

fixes x :: 'a :: linorder

assumes finite S **and** x \in S

shows *is-lowest-element* x S \longleftrightarrow x = Min S

<proof>

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>

lemma *wordinterval-lowest-element-correct-A*:

wordinterval-lowest-element r = Some x \Longrightarrow *is-lowest-element* x (*wordinterval-to-set* r)

<proof>

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>

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)

<proof>

```
fun wordinterval-card :: ('a::len) wordinterval  $\Rightarrow$  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  $\geq$  card (wordinterval-to-set r)  
<proof>
```

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  $\Rightarrow$  'c  $\Rightarrow$  'a)  $\Rightarrow$  'b  $\times$  'c  $\Rightarrow$  'a
```

where

```
  uncurry f a  $\equiv$  (case a of (x,y)  $\Rightarrow$  f x y)
```

```
lemma uncurry-simp[simp]: uncurry f (a,b) = f a b
```

<proof>

```
lemma uncurry-curry-id: uncurry  $\circ$  curry = id curry  $\circ$  uncurry = id
```

<proof>

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

<proof>

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

<proof>

```
lemmas uncurry-splits = uncurry-split uncurry-split-asm
```

```
lemma uncurry-case-stmt: (case x of (a, b)  $\Rightarrow$  f a b) = uncurry f x
```

<proof>

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

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

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

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

lemma *size* ($x::'i::len word$) = $len\text{-}of(TYPE('i))$ *<proof>*

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
 $network\text{-}prefix = (addr \text{ AND } netmask)$
in
 $\{network\text{-}prefix .. network\text{-}prefix \text{ OR } (NOT \text{ netmask})\}$

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} <proof>

lemma *ipset-from-netmask-maxword:*
ipset-from-netmask ip (- 1) = {ip} <proof>

lemma *ipset-from-netmask-zero:*
ipset-from-netmask ip 0 = UNIV <proof>

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

definition *ipset-from-cidr :: 'i::len word ⇒ nat ⇒ 'i::len word set where*
ipset-from-cidr addr pflength ≡
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) ⇒ ipset-from-cidr base len) = uncurry ipset-from-cidr ipcidr <proof>*

lemma *ipset-from-cidr-0: ipset-from-cidr ip 0 = UNIV <proof>*

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} <proof>*

Alternative definition: Considering words as bit lists:

lemma *ipset-from-cidr-bl:*
fixes *addr :: 'i::len word*
shows *ipset-from-cidr addr pflength ≡*
ipset-from-netmask addr (of-bl ((replicate pflength True) @
(replicate ((len-of(TYPE('i))) - pflength))
False)) <proof>

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)}
<proof>

lemma *ipset-from-cidr-alt2:*

fixes $base :: 'i::len\ word$
shows $ipset-from-cidr\ base\ len =$
 $ipset-from-netmask\ base\ (NOT\ (mask\ (LENGTH('i) - len)))$
 $\langle proof \rangle$

In CIDR notation, we cannot express the empty set.

lemma $ipset-from-cidr-not-empty: ipset-from-cidr\ base\ len \neq \{\}$
 $\langle proof \rangle$

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 $ipset-from-cidr\ base\ l = \{base .. base\ OR\ mask\ (LENGTH('i) - l)\}$
 $\langle proof \rangle$

lemma $ipset-from-cidr-large-pfxlen:$
fixes $ip:: 'i::len\ word$
assumes $n \geq LENGTH('i)$
shows $ipset-from-cidr\ ip\ n = \{ip\}$
 $\langle proof \rangle$

lemma $ipset-from-netmask-base-mask-consume:$
fixes $base :: 'i::len\ word$
shows $ipset-from-netmask\ (base\ AND\ NOT\ (mask\ (LENGTH('i) - m)))$
 $(NOT\ (mask\ (LENGTH('i) - m)))$
 $=$
 $ipset-from-netmask\ base\ (NOT\ (mask\ (LENGTH('i) - m)))$
 $\langle proof \rangle$

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

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

The definitions are equal

lemma $ipset-from-cidr-eq-ip-cidr-set:$
fixes $base:: 'i::len\ word$
shows $ipset-from-cidr\ base\ len = ip-cidr-set\ base\ len$
 $\langle proof \rangle$

lemma $ip-cidr-set-change-base: j \in ip-cidr-set\ i\ r \Longrightarrow ip-cidr-set\ j\ r = ip-cidr-set$
 $i\ r$
 $\langle proof \rangle$

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  
  ⟨proof⟩
```

```
lemma wordinterval-to-set (iprange-single ip) = {ip}  
  ⟨proof⟩
```

```
lemma wordinterval-to-set (iprange-interval (ip1, ip2)) = {ip1 .. ip2}  
  ⟨proof⟩
```

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
```

$ipcidr\text{-to}\text{-interval } cidr \equiv (ipcidr\text{-to}\text{-interval}\text{-start } cidr, ipcidr\text{-to}\text{-interval}\text{-end } cidr)$

lemma *ipset-from-cidr-ipcidr-to-interval*:

$ipset\text{-from}\text{-cidr } base \ len = \{ipcidr\text{-to}\text{-interval}\text{-start } (base, len) .. ipcidr\text{-to}\text{-interval}\text{-end } (base, len)\}$
 $\langle proof \rangle$

declare *ipcidr-to-interval-start.simps[simp del] ipcidr-to-interval-end.simps[simp del]*

lemma *ipcidr-to-interval*:

$ipcidr\text{-to}\text{-interval } (base, len) = (s, e) \implies ipset\text{-from}\text{-cidr } base \ len = \{s .. e\}$
 $\langle proof \rangle$

definition *ipcidr-tuple-to-wordinterval* :: $('i::len \text{ word} \times nat) \Rightarrow 'i \text{ wordinterval}$
where

$ipcidr\text{-tuple}\text{-to}\text{-wordinterval } iprng \equiv iprange\text{-interval } (ipcidr\text{-to}\text{-interval } iprng)$

lemma *wordinterval-to-set-ipcidr-tuple-to-wordinterval*:

$wordinterval\text{-to}\text{-set } (ipcidr\text{-tuple}\text{-to}\text{-wordinterval } (b, m)) = ipset\text{-from}\text{-cidr } b \ m$
 $\langle proof \rangle$

lemma *wordinterval-to-set-ipcidr-tuple-to-wordinterval-uncurry*:

$wordinterval\text{-to}\text{-set } (ipcidr\text{-tuple}\text{-to}\text{-wordinterval } ipcidr) = uncurry \ ipset\text{-from}\text{-cidr } ipcidr$
 $\langle proof \rangle$

definition *ipcidr-union-set* :: $('i::len \text{ word} \times nat) \text{ set} \Rightarrow ('i \text{ word}) \text{ set}$ **where**

$ipcidr\text{-union}\text{-set } ips \equiv \bigcup (base, len) \in ips. ipset\text{-from}\text{-cidr } base \ len$

lemma *ipcidr-union-set-uncurry*:

$ipcidr\text{-union}\text{-set } ips = (\bigcup ipcidr \in ips. uncurry \ ipset\text{-from}\text{-cidr } ipcidr)$
 $\langle proof \rangle$

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 $i :: ('a :: len) \text{ word}$
assumes $r2 \leq r1 \ i \ \&\& \ \sim\sim (mask \ r2) = x \ \&\& \ \sim\sim (mask \ r2)$

shows $i \&\& \sim\sim (\text{mask } r1) = x \&\& \sim\sim (\text{mask } r1)$
 ⟨proof⟩

lemma *ip-cidr-set-less*:

fixes $i :: 'i::\text{len word}$

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

⟨proof⟩ **lemma** *ip-cidr-set-intersect-subset-helper*:

fixes $i1 \ r1 \ i2 \ r2$

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

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

⟨proof⟩

lemma *ip-cidr-set-notsubset-empty-inter*:

$\neg \text{ip-cidr-set } i1 \ r1 \subseteq \text{ip-cidr-set } i2 \ r2 \implies$

$\neg \text{ip-cidr-set } i2 \ r2 \subseteq \text{ip-cidr-set } i1 \ r1 \implies$

$\text{ip-cidr-set } i1 \ r1 \cap \text{ip-cidr-set } i2 \ r2 = \{\}$

⟨proof⟩

end

lemma *ip-cidr-intersect*:

$\neg \text{ipset-from-cidr } b2 \ m2 \subseteq \text{ipset-from-cidr } b1 \ m1 \implies$

$\neg \text{ipset-from-cidr } b1 \ m1 \subseteq \text{ipset-from-cidr } b2 \ m2 \implies$

$\text{ipset-from-cidr } b1 \ m1 \cap \text{ipset-from-cidr } b2 \ m2 = \{\}$

⟨proof⟩

Computing the intersection of two IP address ranges in CIDR notation

fun *ipcidr-conjunct* :: $('i::\text{len word} \times \text{nat}) \Rightarrow ('i \ \text{word} \times \text{nat}) \Rightarrow ('i \ \text{word} \times \text{nat})$

option where

ipcidr-conjunct ($\text{base1}, m1$) ($\text{base2}, m2$) = (

if

$\text{ipset-from-cidr } \text{base1} \ m1 \cap \text{ipset-from-cidr } \text{base2} \ m2 = \{\}$

then

None

else if

$\text{ipset-from-cidr } \text{base1} \ m1 \subseteq \text{ipset-from-cidr } \text{base2} \ m2$

then

Some ($\text{base1}, m1$)

else

Some ($\text{base2}, m2$)

)

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

lemma *ipcidr-conjunct-any*: $\text{ipcidr-conjunct } a \ (x,0) \neq \text{None } \text{ipcidr-conjunct } (y,0)$

$b \neq \text{None}$

⟨proof⟩

lemma *ipcidr-conjunct-correct*: $(\text{case } \text{ipcidr-conjunct } (b1, m1) (b2, m2))$

$$\begin{aligned} & \text{of } \text{Some } (bx, mx) \Rightarrow \text{ipset-from-cidr } bx \ mx \\ & | \text{None} \Rightarrow \{\} = \\ & (\text{ipset-from-cidr } b1 \ m1) \cap (\text{ipset-from-cidr } b2 \ m2) \end{aligned}$$

<proof>

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*
)
<proof>

lemma *ipcidr-conjunct* (*0::32 word,0*) (*8,1*) = *Some* (*8, 1*) *<proof>*

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*) \longleftrightarrow
 (*base AND netmask*) ≤ *addr* ∧ *addr* ≤ (*base AND netmask*) OR (*NOT*
netmask)
<proof>

lemma *addr-in-ipset-from-cidr-code*[*code-unfold*]:
 (*addr::'i::len word*) ∈ (*ipset-from-cidr pre len*) \longleftrightarrow
 (*pre AND* ((*mask len*) << (*LENGTH('i) - len*))) ≤ *addr* ∧
 addr ≤ *pre* OR (*mask* (*LENGTH('i) - len*))
<proof>

end

end

theory *IPv4*

imports *IP-Address*

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]:
 $\langle a \ \&\& \ 4294967295 = a \rangle$ **for** $a :: \text{ipv4addr}$
 <proof>

Conversion between natural numbers and IPv4 addresses

definition *nat-of-ipv4addr* :: *ipv4addr* \Rightarrow *nat* **where**
nat-of-ipv4addr $a = \text{unat } a$

definition *ipv4addr-of-nat* :: *nat* \Rightarrow *ipv4addr* **where**
ipv4addr-of-nat $n = \text{of-nat } n$

The maximum IPv4 address

definition *max-ipv4-addr* :: *ipv4addr* **where**
max-ipv4-addr $\equiv \text{ipv4addr-of-nat } ((2^{32}) - 1)$

lemma *max-ipv4-addr-number*: *max-ipv4-addr* = 4294967295
 <proof>

lemma *max-ipv4-addr* = 0b11111111111111111111111111111111
 <proof>

lemma *max-ipv4-addr-max-word*: *max-ipv4-addr* = - 1
 <proof>

lemma *max-ipv4-addr-max*[simp]: $\forall a. a \leq \text{max-ipv4-addr}$
 <proof>

lemma *UNIV-ipv4addrset*: $\text{UNIV} = \{0 .. \text{max-ipv4-addr}\}$
 <proof>

identity functions

lemma *nat-of-ipv4addr-ipv4addr-of-nat-mod*: *nat-of-ipv4addr* (*ipv4addr-of-nat* n)
 = $n \bmod 2^{32}$
 <proof>

lemma *nat-of-ipv4addr-ipv4addr-of-nat*:
 $\llbracket n \leq \text{nat-of-ipv4addr } \text{max-ipv4-addr} \rrbracket \Longrightarrow \text{nat-of-ipv4addr } (\text{ipv4addr-of-nat } n)$
 = n
 <proof>

lemma *ipv4addr-of-nat-nat-of-ipv4addr*: *ipv4addr-of-nat* (*nat-of-ipv4addr* addr)
 = addr
 <proof>

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
  <proof>

```

```

lemma dotdecimal-of-ipv4addr 3232235521 = (192, 168, 0, 1)
  <proof>

```

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>

```

```

lemma size-ipv4addr: size (x::ipv4addr) = 32 <proof>

```

```

lemma dotdecimal-of-ipv4addr-ipv4addr-of-dotdecimal:
  [ a < 256; b < 256; c < 256; d < 256 ] ⇒
  dotdecimal-of-ipv4addr (ipv4addr-of-dotdecimal (a,b,c,d)) = (a,b,c,d)
  <proof>

```

```

lemma ipv4addr-of-dotdecimal-dotdecimal-of-ipv4addr:
  (ipv4addr-of-dotdecimal (dotdecimal-of-ipv4addr ip)) = ip
  <proof>

```

```

lemma ipv4addr-of-dotdecimal-eqE:
  [ 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
  ] ⇒
  a = e ∧ b = f ∧ c = g ∧ d = h
  <proof>

```

4.2 IP Ranges: Examples

lemma $(UNIV :: ipv4addr\ set) = \{0 .. max-ipv4-addr\}$ $\langle proof \rangle$

lemma $(42::ipv4addr) \in UNIV$ $\langle proof \rangle$

lemma $ipset-from-netmask\ (ipv4addr-of-dotdecimal\ (192,168,0,42))\ (ipv4addr-of-dotdecimal\ (255,255,0,0)) =$

$\{ipv4addr-of-dotdecimal\ (192,168,0,0) .. ipv4addr-of-dotdecimal\ (192,168,255,255)\}$
 $\langle proof \rangle$

lemma $ipset-from-netmask\ (ipv4addr-of-dotdecimal\ (192,168,0,42))\ (ipv4addr-of-dotdecimal\ (0,0,0,0)) = UNIV$

$\langle proof \rangle$

192.168.0.0/24

lemma **fixes** $addr :: ipv4addr$

shows $ipset-from-cidr\ addr\ pflength =$

$ipset-from-netmask\ addr\ ((mask\ pflength) << (32 - pflength))$

$\langle proof \rangle$

lemma $ipset-from-cidr\ (ipv4addr-of-dotdecimal\ (192,168,0,42))\ 16 =$

$\{ipv4addr-of-dotdecimal\ (192,168,0,0) .. ipv4addr-of-dotdecimal\ (192,168,255,255)\}$

$\langle proof \rangle$

lemma $ip \in (ipset-from-cidr\ (ipv4addr-of-dotdecimal\ (0, 0, 0, 0))\ 0)$

$\langle proof \rangle$

lemma $ipv4set-from-cidr-32$: **fixes** $addr :: ipv4addr$

shows $ipset-from-cidr\ addr\ 32 = \{addr\}$

$\langle proof \rangle$

lemma **fixes** $pre :: ipv4addr$

shows $ipset-from-cidr\ pre\ len = \{(pre\ AND\ ((mask\ len) << (32 - len))) .. pre\ OR\ (mask\ (32 - len))\}$

$\langle proof \rangle$

making element check executable

lemma $addr-in-ipv4set-from-netmask-code[code-unfold]$:

fixes $addr :: ipv4addr$

shows $addr \in (ipset-from-netmask\ base\ netmask) \longleftrightarrow$

$(base\ AND\ netmask) \leq addr \wedge addr \leq (base\ AND\ netmask)\ OR\ (NOT\ netmask)$

$\langle proof \rangle$

lemma $addr-in-ipv4set-from-cidr-code[code-unfold]$:

fixes $addr :: ipv4addr$

shows $addr \in (ipset-from-cidr\ pre\ len) \longleftrightarrow$

$(pre\ AND\ ((mask\ len) << (32 - len))) \leq addr \wedge addr \leq pre\ OR\ (mask\ (32 - len))$

<proof>

lemma *ipv4addr-of-dotdecimal* (192,168,42,8) ∈ (*ipset-from-cidr* (*ipv4addr-of-dotdecimal* (192,168,0,0)) 16)

<proof>

definition *ipv4range-UNIV* :: 32 *wordinterval* **where** *ipv4range-UNIV* ≡ *wordinterval-UNIV*

lemma *ipv4range-UNIV-set-eq*: *wordinterval-to-set* *ipv4range-UNIV* = *UNIV*

<proof>

thm *iffD1*[*OF wordinterval-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)*))

<proof>

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* ⇒ *nat* **where**

nat-of-ipv6addr a = *unat a*

definition *ipv6addr-of-nat* :: *nat* ⇒ *ipv6addr* **where**

ipv6addr-of-nat n = *of-nat n*

lemma *ipv6addr-of-nat* $n = \text{word-of-int } (\text{int } n)$
<proof>

The maximum IPv6 address

definition *max-ipv6-addr* :: *ipv6addr* **where**
max-ipv6-addr $\equiv \text{ipv6addr-of-nat } ((2^{128}) - 1)$

lemma *max-ipv6-addr-number*: *max-ipv6-addr* = *0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF*
<proof>

lemma *max-ipv6-addr* = *340282366920938463463374607431768211455*
<proof>

lemma *max-ipv6-addr-max-word*: *max-ipv6-addr* = *- 1*
<proof>

lemma *max-ipv6-addr-max*: $\forall a. a \leq \text{max-ipv6-addr}$
<proof>

lemma *UNIV-ipv6addrset*: *UNIV* = $\{0 .. \text{max-ipv6-addr}\}$
<proof>

identity functions

lemma *nat-of-ipv6addr-ipv6addr-of-nat-mod*: *nat-of-ipv6addr* (*ipv6addr-of-nat* n)
= $n \bmod 2^{128}$
<proof>

lemma *nat-of-ipv6addr-ipv6addr-of-nat*:
 $n \leq \text{nat-of-ipv6addr } \text{max-ipv6-addr} \implies \text{nat-of-ipv6addr } (\text{ipv6addr-of-nat } n) = n$
<proof>

lemma *ipv6addr-of-nat-nat-of-ipv6addr*: *ipv6addr-of-nat* (*nat-of-ipv6addr* *addr*)
= *addr*
<proof>

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.

<i>IPv6AddrCompressed1-0</i>	<i>unit</i>
<i>IPv6AddrCompressed1-1</i>	<i>unit 16 word</i>
<i>IPv6AddrCompressed1-2</i>	<i>unit 16 word 16 word</i>
<i>IPv6AddrCompressed1-3</i>	<i>unit 16 word 16 word 16 word</i>
<i>IPv6AddrCompressed1-4</i>	<i>unit 16 word 16 word 16 word 16 word</i>
<i>IPv6AddrCompressed1-5</i>	<i>unit 16 word 16 word 16 word 16 word 16 word</i>
<i>IPv6AddrCompressed1-6</i>	<i>unit 16 word 16 word 16 word 16 word 16 word 16 word</i>
<i>IPv6AddrCompressed1-7</i>	<i>unit 16 word 16 word 16 word 16 word 16 word 16 word 16 word</i>
<i>16 word</i>	
<i>IPv6AddrCompressed2-1</i>	<i>16 word unit</i>
<i>IPv6AddrCompressed2-2</i>	<i>16 word unit 16 word</i>
<i>IPv6AddrCompressed2-3</i>	<i>16 word unit 16 word 16 word</i>
<i>IPv6AddrCompressed2-4</i>	<i>16 word unit 16 word 16 word 16 word</i>
<i>IPv6AddrCompressed2-5</i>	<i>16 word unit 16 word 16 word 16 word 16 word</i>
<i>IPv6AddrCompressed2-6</i>	<i>16 word unit 16 word 16 word 16 word 16 word 16 word</i>
<i>IPv6AddrCompressed2-7</i>	<i>16 word unit 16 word 16 word 16 word 16 word 16 word 16 word</i>
<i>16 word</i>	
<i>IPv6AddrCompressed3-2</i>	<i>16 word 16 word unit</i>
<i>IPv6AddrCompressed3-3</i>	<i>16 word 16 word unit 16 word</i>
<i>IPv6AddrCompressed3-4</i>	<i>16 word 16 word unit 16 word 16 word</i>
<i>IPv6AddrCompressed3-5</i>	<i>16 word 16 word unit 16 word 16 word 16 word</i>
<i>IPv6AddrCompressed3-6</i>	<i>16 word 16 word unit 16 word 16 word 16 word 16 word</i>
<i>IPv6AddrCompressed3-7</i>	<i>16 word 16 word unit 16 word 16 word 16 word 16 word 16 word</i>
<i>16 word</i>	
<i>IPv6AddrCompressed4-3</i>	<i>16 word 16 word 16 word unit</i>

| 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

| 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

| 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

| IPv6AddrCompressed7-6 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
 unit 16 word

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

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

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

 | [Some a, None] ⇒ Some (IPv6AddrCompressed2-1 a ())
 | [Some a, None, Some b] ⇒ Some (IPv6AddrCompressed2-2 a () b)
 | [Some a, None, Some b, Some c] ⇒ Some (IPv6AddrCompressed2-3 a () b c)
 | [Some a, None, Some b, Some c, Some d] ⇒ Some (IPv6AddrCompressed2-4
 a () b c d)

| [Some a, None, Some b, Some c, Some d, Some e] ⇒ Some (IPv6AddrCompressed2-5
 a () b c d e)
 | [Some a, None, Some b, Some c, Some d, Some e, Some f] ⇒ Some (IPv6AddrCompressed2-6
 a () b c d e f)
 | [Some a, None, Some b, Some c, Some d, Some e, Some f, Some g] ⇒ Some
 (IPv6AddrCompressed2-7 a () b c d e f g)

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

```

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

$| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed4-3\ a\ b\ c\ ()) =$
 $[Some\ a,\ Some\ b,\ Some\ c,\ None]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed4-4\ a\ b\ c\ ()\ d) =$
 $[Some\ a,\ Some\ b,\ Some\ c,\ None,\ Some\ d]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed4-5\ a\ b\ c\ ()\ d\ e) =$
 $[Some\ a,\ Some\ b,\ Some\ c,\ None,\ Some\ d,\ Some\ e]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed4-6\ a\ b\ c\ ()\ d\ e\ f) =$
 $[Some\ a,\ Some\ b,\ Some\ c,\ None,\ Some\ d,\ Some\ e,$
 $Some\ f]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed4-7\ a\ b\ c\ ()\ d\ e\ f\ g)$
 $=$
 $[Some\ a,\ Some\ b,\ Some\ c,\ None,\ Some\ d,\ Some\ e,$
 $Some\ f,\ Some\ g]$

$| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed5-4\ a\ b\ c\ d\ ()) =$
 $[Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ None]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed5-5\ a\ b\ c\ d\ ()\ e) =$
 $[Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ None,\ Some\ e]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed5-6\ a\ b\ c\ d\ ()\ e\ f) =$
 $[Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ None,\ Some\ e,$
 $Some\ f]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed5-7\ a\ b\ c\ d\ ()\ e\ f\ g)$
 $=$
 $[Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ None,\ Some\ e,$
 $Some\ f,\ Some\ g]$

$| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed6-5\ a\ b\ c\ d\ e\ ()) =$
 $[Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ Some\ e,\ None]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed6-6\ a\ b\ c\ d\ e\ ()\ f) =$
 $[Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ Some\ e,\ None,$
 $Some\ f]$
 $| ipv6addr-syntax-compressed-to-list\ (IPv6AddrCompressed6-7\ a\ b\ c\ d\ e\ ()\ f\ g)$
 $=$
 $[Some\ a,\ Some\ b,\ Some\ c,\ Some\ d,\ Some\ e,\ None,$
 $Some\ f,\ Some\ g]$

$| \text{ipv6addr-syntax-compressed-to-list } (IPv6AddrCompressed7-6 \ a \ b \ c \ d \ e \ f \ ()) =$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ Some \ d, \ Some \ e, \ Some \ f,$
 $None]$
 $| \text{ipv6addr-syntax-compressed-to-list } (IPv6AddrCompressed7-7 \ a \ b \ c \ d \ e \ f \ () \ g)$
 $=$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ Some \ d, \ Some \ e, \ Some \ f,$
 $None, \ Some \ g]$

 $| \text{ipv6addr-syntax-compressed-to-list } (IPv6AddrCompressed8-7 \ a \ b \ c \ d \ e \ f \ g \ ())$
 $=$
 $\quad [Some \ a, \ Some \ b, \ Some \ c, \ Some \ d, \ Some \ e, \ Some \ f,$
 $Some \ g, \ None]$

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

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

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

parse-ipv6-address-compressed (ipv6addr-syntax-compressed-to-list (ipv6-syntax))
 $=$ *Some ipv6-syntax*
<proof>

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

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

obtains

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

 $a \ \text{where} \ as = [Some \ a, \ None] \ \text{ipv6} = (IPv6AddrCompressed2-1 \ a \ ()) \ |$
 $a \ b \ \text{where} \ as = [Some \ a, \ None, \ Some \ b] \ \text{ipv6} = (IPv6AddrCompressed2-2 \ a \ ()$
 $b) \ |$
 $a \ b \ c \ \text{where} \ as = [Some \ a, \ None, \ Some \ b, \ Some \ c] \ \text{ipv6} = (IPv6AddrCompressed2-3$
 $a \ () \ b \ c) \ |$
 $a \ b \ c \ d \ \text{where} \ as = [Some \ a, \ None, \ Some \ b, \ Some \ c, \ Some \ d] \ \text{ipv6} =$

(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 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) |
a b c d e **where** *as = [Some a, Some b, None, Some c, Some d, Some e]* *ipv6*
= (IPv6AddrCompressed3-5 a b () c d e) |
a b c d e f **where** *as = [Some a, Some b, None, Some c, Some d, Some e,*
Some f] *ipv6 = (IPv6AddrCompressed3-6 a b () c d e f) |*
a b c d e f g **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) |*

a b c **where** *as = [Some a, Some b, Some c, None]* *ipv6 = (IPv6AddrCompressed4-3*
a b c ()) |
a b c d **where** *as = [Some a, Some b, Some c, None, Some d]* *ipv6 =*
(IPv6AddrCompressed4-4 a b c () d) |
a b c d e **where** *as = [Some a, Some b, Some c, None, Some d, Some e]* *ipv6*
= (IPv6AddrCompressed4-5 a b c () d e) |
a b c d e f **where** *as = [Some a, Some b, Some c, None, Some d, Some e,*
Some f] *ipv6 = (IPv6AddrCompressed4-6 a b c () d e f) |*
a b c d e f g **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) |*

a b c d **where** *as = [Some a, Some b, Some c, Some d, None]* *ipv6 =*
(IPv6AddrCompressed5-4 a b c d ()) |
a b c d e **where** *as = [Some a, Some b, Some c, Some d, None, Some e]* *ipv6*
= (IPv6AddrCompressed5-5 a b c d () e) |
a b c d e f **where** *as = [Some a, Some b, Some c, Some d, None, Some e,*
Some f] *ipv6 = (IPv6AddrCompressed5-6 a b c d () e f) |*
a b c d e f g **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) |*

a b c d e **where** *as = [Some a, Some b, Some c, Some d, Some e, None]* *ipv6*
= (IPv6AddrCompressed6-5 a b c d e ()) |
a b c d e f **where** *as = [Some a, Some b, Some c, Some d, Some e, None,*
Some f] *ipv6 = (IPv6AddrCompressed6-6 a b c d e () f) |*
a b c d e f g **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) |*

a b c d e f **where** *as = [Some a, Some b, Some c, Some d, Some e, Some f,*

None] *ipv6* = (*IPv6AddrCompressed7-6* *a b c d e f* ()) |
a b c d e f g **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*) |

a b c d e f g **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* ())
 <proof>

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>

Valid IPv6 compressed notation:

- at most one omission
- at most 7 pieces

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

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 <proof>

```

```

lemma ipv6preferred-to-int (IPv6AddrPreferred 0xFF01 0x0 0x0 0x0 0x0 0x0 0x0
0x101) =
  338958331222012082418099330867817087233 <proof>

```

```

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 <proof>

```

```

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
  (0xFFFF0000::ipv6addr) = (mask 16) << 16
  (0xFFFF::ipv6addr) = (mask 16)
  <proof>

```

Correctness: round trip property one

lemma *ipv6preferred-to-int-int-to-ipv6preferred*:
ipv6preferred-to-int (int-to-ipv6preferred ip) = ip
<proof>

Correctness: round trip property two

lemma *int-to-ipv6preferred-ipv6preferred-to-int*: *int-to-ipv6preferred (ipv6preferred-to-int ip) = ip*
<proof>

compressed to preferred format

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

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

| ipv6addr-c2p (IPv6AddrCompressed3-2 a b ()) = IPv6AddrPreferred a b 0 0 0 0 0 0 0 0
0 0 0
| ipv6addr-c2p (IPv6AddrCompressed3-3 a b () h) = IPv6AddrPreferred a b 0 0 0 0 0 0 h
0 0 0 h

| *ipv6addr-c2p* (*IPv6AddrCompressed3-4* *a b () g h*) = *IPv6AddrPreferred a b 0 0 0 0 g h*
 | *ipv6addr-c2p* (*IPv6AddrCompressed3-5* *a b () f g h*) = *IPv6AddrPreferred a b 0 0 0 f g h*
 | *ipv6addr-c2p* (*IPv6AddrCompressed3-6* *a b () e f g h*) = *IPv6AddrPreferred a b 0 0 e f g h*
 | *ipv6addr-c2p* (*IPv6AddrCompressed3-7* *a b () d e f g h*) = *IPv6AddrPreferred a b 0 d e f g h*

| *ipv6addr-c2p* (*IPv6AddrCompressed4-3* *a b c ()*) = *IPv6AddrPreferred a b c 0 0 0 0 0*
 | *ipv6addr-c2p* (*IPv6AddrCompressed4-4* *a b c () h*) = *IPv6AddrPreferred a b c 0 0 0 0 h*
 | *ipv6addr-c2p* (*IPv6AddrCompressed4-5* *a b c () g h*) = *IPv6AddrPreferred a b c 0 0 0 g h*
 | *ipv6addr-c2p* (*IPv6AddrCompressed4-6* *a b c () f g h*) = *IPv6AddrPreferred a b c 0 0 f g h*
 | *ipv6addr-c2p* (*IPv6AddrCompressed4-7* *a b c () e f g h*) = *IPv6AddrPreferred a b c 0 e f g h*

| *ipv6addr-c2p* (*IPv6AddrCompressed5-4* *a b c d ()*) = *IPv6AddrPreferred a b c d 0 0 0 0*
 | *ipv6addr-c2p* (*IPv6AddrCompressed5-5* *a b c d () h*) = *IPv6AddrPreferred a b c d 0 0 0 h*
 | *ipv6addr-c2p* (*IPv6AddrCompressed5-6* *a b c d () g h*) = *IPv6AddrPreferred a b c d 0 0 g h*
 | *ipv6addr-c2p* (*IPv6AddrCompressed5-7* *a b c d () f g h*) = *IPv6AddrPreferred a b c d 0 f g h*

| *ipv6addr-c2p* (*IPv6AddrCompressed6-5* *a b c d e ()*) = *IPv6AddrPreferred a b c d e 0 0 0*
 | *ipv6addr-c2p* (*IPv6AddrCompressed6-6* *a b c d e () h*) = *IPv6AddrPreferred a b c d e 0 0 h*
 | *ipv6addr-c2p* (*IPv6AddrCompressed6-7* *a b c d e () g h*) = *IPv6AddrPreferred a b c d e 0 g h*

| *ipv6addr-c2p* (*IPv6AddrCompressed7-6* *a b c d e f ()*) = *IPv6AddrPreferred a b c d e f 0 0*
 | *ipv6addr-c2p* (*IPv6AddrCompressed7-7* *a b c d e f () h*) = *IPv6AddrPreferred a b c d e f 0 h*

| *ipv6addr-c2p* (*IPv6AddrCompressed8-7* *a b c d e f g ()*) = *IPv6AddrPreferred a b c d e f g 0*

definition *ipv6-unparsed-compressed-to-preferred* :: ((16 word) option) list ⇒ *ipv6addr-syntax*
 option **where**
 ipv6-unparsed-compressed-to-preferred *ls* = (
 if

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 group-by-zeros :: 16 word list  $\Rightarrow$  16 word list list where  
  group-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))
<proof>

```

termination goup-by-zeros

```

<proof> lemma goup-by-zeros [0,1,2,3,0,0,0,3,4,0,0,0,2,0,0,2,0,3,0] =
  [[0], [1], [2], [3], [0, 0, 0, 0], [3], [4], [0, 0, 0], [2], [0, 0], [2], [0], [3], [0]]
<proof> lemma concat (goup-by-zeros ls) = ls
<proof> lemma []  $\notin$  set (goup-by-zeros ls)
<proof> 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

```

<proof> lemma a  $\notin$  set ls  $\implies$  List-replace1 a b ls = ls
<proof> lemma a  $\in$  set ls  $\implies$  b  $\in$  set (List-replace1 a b ls)
<proof> fun List-explode :: 'a list list  $\Rightarrow$  ('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]

<proof> lemma List-explode-def:

```

List-explode xss = concat (map ( $\lambda xs. \text{if } xs = [] \text{ then } [None] \text{ else } \text{map Some } xs$ ) xss)

```

<proof> lemma List-explode-no-empty: [] \notin set xss \implies List-explode xss = map Some (concat xss)

```

<proof> lemma List-explode-replace1: []  $\notin$  set xss  $\implies$  foo  $\in$  set xss  $\implies$ 
  List-explode (List-replace1 foo [] xss) =
    map Some (concat (takeWhile ( $\lambda xs. xs \neq \text{foo}$ ) xss)) @ [None] @
    map Some (concat (tl (dropWhile ( $\lambda xs. xs \neq \text{foo}$ ) xss)))

```

<proof>

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

```

ipv6-preferred-to-compressed (IPv6AddrPreferred a b c d e f g h) = (
  let lss = goup-by-zeros [a,b,c,d,e,f,g,h];
      max-zero-seq = foldr ( $\lambda xs. \text{max } (\text{length } xs)$ ) 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*: $\text{foldr } (\lambda x s. \text{max } (\text{length } xs)) \text{ lss } n = \text{fold max } (\text{map length lss}) n$
 ⟨proof⟩ **lemma** *List-explode-goup-by-zeros*: $\text{List-explode } (\text{goup-by-zeros } xs) = \text{map Some } xs$
 ⟨proof⟩ **definition** *max-zero-streak* $xs \equiv \text{foldr } (\lambda x s. \text{max } (\text{length } xs)) (\text{goup-by-zeros } xs) 0$

private lemma *max-zero-streak-def2*: $\text{max-zero-streak } xs = \text{fold max } (\text{map length } (\text{goup-by-zeros } xs)) 0$
 ⟨proof⟩ **lemma** *ipv6-preferred-to-compressed-pull-out-if*:
 $\text{ipv6-preferred-to-compressed } (\text{IPv6AddrPreferred } a \ b \ c \ d \ e \ f \ g \ h) =$
 if $\text{max-zero-streak } [a, b, c, d, e, f, g, h] > 1$ then
 $\text{List-explode } (\text{List-replace1 } (\text{replicate } (\text{max-zero-streak } [a, b, c, d, e, f, g, h]) 0) []$
 $(\text{goup-by-zeros } [a, b, c, d, e, f, g, h]))$
 else
 $\text{map Some } [a, b, c, d, e, f, g, h]$
)
 ⟨proof⟩ **lemma** *ipv6-preferred-to-compressed* $(\text{IPv6AddrPreferred } 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0) = [\text{None}]$ ⟨proof⟩ **lemma** *ipv6-preferred-to-compressed* $(\text{IPv6AddrPreferred } 0x2001 \ 0xDB8 \ 0 \ 0 \ 8 \ 0x800 \ 0x200C \ 0x417A) =$
 $[\text{Some } 0x2001, \text{Some } 0xDB8, \text{None}, \text{Some } 8, \text{Some } 0x800, \text{Some } 0x200C, \text{Some } 0x417A]$ ⟨proof⟩ **lemma** *ipv6-preferred-to-compressed*
 $(\text{IPv6AddrPreferred } 0x2001 \ 0xDB8 \ 0 \ 3 \ 8 \ 0x800 \ 0x200C \ 0x417A) =$
 $[\text{Some } 0x2001, \text{Some } 0xDB8, \text{Some } 0, \text{Some } 3, \text{Some } 8, \text{Some } 0x800,$
 $\text{Some } 0x200C, \text{Some } 0x417A]$ ⟨proof⟩

lemma *ipv6-preferred-to-compressed-RFC-4291-format*:
 $\text{ipv6-preferred-to-compressed } ip = as \implies$
 $\text{length } (\text{filter } (\lambda p. p = \text{None}) as) = 0 \wedge \text{length } as = 8$
 \vee
 $\text{length } (\text{filter } (\lambda p. p = \text{None}) as) = 1 \wedge \text{length } (\text{filter } (\lambda p. p \neq \text{None}) as)$
 ≤ 7
 ⟨proof⟩ **lemma** *ipv6-preferred-to-compressed* $(\text{IPv6AddrPreferred } a \ b \ c \ d \ e \ f \ g \ h)$
 $= \text{None} \# xs \implies$
 $xs = \text{map Some } (\text{dropWhile } (\lambda x. x = 0) [a, b, c, d, e, f, g, h])$
 ⟨proof⟩

lemma *ipv6-preferred-to-compressed*:
assumes *ipv6-unparsed-compressed-to-preferred* $(\text{ipv6-preferred-to-compressed } ip) = \text{Some } ip'$
shows $ip = ip'$
 ⟨proof⟩

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) <proof>

```

When zeroing all least significant bits which exceed the *pfm-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>

```

```

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 \times nat* directly.

lemma *valid-prefix-00*: *valid-prefix* (*PrefixMatch* 0 0) \langle *proof* \rangle

definition *prefix-match-to-CIDR* :: ('i::len) *prefix-match* \Rightarrow ('i word \times nat) **where**
prefix-match-to-CIDR pfx \equiv (*pfxm-prefix* pfx, *pfxm-length* pfx)

lemma *prefix-match-to-CIDR-def2*: *prefix-match-to-CIDR* = (λ pfx. (*pfxm-prefix* pfx, *pfxm-length* pfx))
 \langle *proof* \rangle

definition *prefix-match-dtor* m \equiv (*case* m of *PrefixMatch* p l \Rightarrow (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 $a \leq b \iff$ (*if* *pfxm-length* a = *pfxm-length* b
then *pfxm-prefix* a \leq *pfxm-prefix* b
else *pfxm-length* a $>$ *pfxm-length* b)

definition $a < b \iff$ (a \neq b \wedge
(*if* *pfxm-length* a = *pfxm-length* b
then *pfxm-prefix* a \leq *pfxm-prefix* b
else *pfxm-length* a $>$ *pfxm-length* b))

instance

\langle *proof* \rangle

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]
 \langle *proof* \rangle

context

includes *bit-operations-syntax*

begin

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

\langle *proof* \rangle **lemma** *valid-prefix-alt*: **fixes** p::'a::len *prefix-match*

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

\langle *proof* \rangle

6.1 Address Semantics

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

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

prefix-match-semantics m a \equiv *pfm-prefix* m = NOT (*pfm-mask* m) AND a

lemma *same-length-prefixes-distinct*: *valid-prefix* pfx1 \implies *valid-prefix* pfx2 \implies
pfx1 \neq pfx2 \implies *pfm-length* pfx1 = *pfm-length* pfx2 \implies *prefix-match-semantics*
pfx1 w \implies *prefix-match-semantics* pfx2 w \implies False

<proof>

6.2 Relation between prefix and set

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

prefix-to-wordset pfx = {*pfm-prefix* pfx .. *pfm-prefix* pfx OR *pfm-mask* pfx}

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

<proof>

lemma *zero-prefix-match-all*:

valid-prefix m \implies *pfm-length* m = 0 \implies *prefix-match-semantics* m ip

<proof>

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

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

<proof>

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

<proof> **lemma** *valid-prefix-ipset-from-netmask-ipset-from-cidr*:

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

<proof>

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

assumes *valid-prefix* pfx

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

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

<proof>

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* (*pfm-prefix* pfx) (*pfm-length* pfx)

<proof>

lemma *prefix-to-wordset-ipset-from-cidr*:
assumes *valid-prefix* (*px::'a::len prefix-match*)
shows *prefix-to-wordset pfx = ipset-from-cidr (pfxm-prefix pfx) (pfxm-length pfx)*
 <proof>

definition *prefix-to-wordinterval* :: *'a::len prefix-match* \Rightarrow *'a wordinterval* **where**
prefix-to-wordinterval pfx \equiv *WordInterval (pfxm-prefix pfx) (pfxm-prefix pfx OR pfxm-mask pfx)*

lemma *prefix-to-wordinterval-set-eq[simp]*:
wordinterval-to-set (prefix-to-wordinterval pfx) = prefix-to-wordset pfx
 <proof>

lemma *prefix-to-wordinterval-def2*:
prefix-to-wordinterval pfx =
iprange-interval ((pfxm-prefix pfx), (pfxm-prefix pfx OR pfxm-mask pfx))
 <proof>

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)
 <proof>

end

lemma *prefix-never-empty*:
fixes *d::'a::len prefix-match*
shows \neg *wordinterval-empty (prefix-to-wordinterval d)*
 <proof>

Getting a lowest element

lemma *ipset-from-cidr-lowest: a \in ipset-from-cidr a n*
 <proof>

lemma *valid-prefix (PrefixMatch a n) \implies is-lowest-element a (ipset-from-cidr a n)*
 <proof>

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 ∘ nat) [1 .. 4] = ([1, 2, 3, 4]:: 32 word list)* *<proof>*
definition *ipv4addr-upto :: 32 word ⇒ 32 word ⇒ 32 word list* **where**
ipv4addr-upto i j ≡ map (of-nat ∘ nat) [int (unat i) .. int (unat j)]
private lemma *ipv4addr-upto: set (ipv4addr-upto i j) = {i .. j}*
<proof>

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*
<proof> **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]* *<proof>* **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 ⇒ 'a prefix-match option × 'a wordinterval **where**
wordinterval-CIDR-split1 r ≡ (
let ma = wordinterval-lowest-element r in
case ma of
None ⇒ (None, r) |
Some a ⇒ (case largest-contained-prefix a r of
None ⇒ (None, r) |
Some m ⇒ (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 ⇒*

largest-contained-prefix a r ≠ None
 ⟨proof⟩ **lemma** *r-split1-not-none*: **fixes** *r:: 'a::len wordinterval*
shows \neg *wordinterval-empty r* \implies *fst (wordinterval-CIDR-split1 r) ≠ None*
 ⟨proof⟩ **lemma** *largest-contained-prefix-subset*:
largest-contained-prefix a r = Some p \implies *wordinterval-to-set (prefix-to-wordinterval*
p) ⊆ wordinterval-to-set r
 ⟨proof⟩ **lemma** *wordinterval-CIDR-split1-snd*: *wordinterval-CIDR-split1 r = (Some*
s, u) \implies *u = wordinterval-setminus r (prefix-to-wordinterval s)*
 ⟨proof⟩ **lemma** *largest-contained-prefix-subset-s1D*:
wordinterval-CIDR-split1 r = (Some s, u) \implies *wordinterval-to-set (prefix-to-wordinterval*
s) ⊆ wordinterval-to-set r
 ⟨proof⟩ **theorem** *wordinterval-CIDR-split1-preserve*: **fixes** *r:: 'a::len wordinterval*
shows *wordinterval-CIDR-split1 r = (Some s, u)* \implies *wordinterval-eq (wordinterval-union*
(prefix-to-wordinterval s) u) r
 ⟨proof⟩ **lemma** *wordinterval-CIDR-split1-some-r-ne*:
wordinterval-CIDR-split1 r = (Some s, u) \implies \neg *wordinterval-empty r*
 ⟨proof⟩ **lemma** *wordinterval-CIDR-split1-distinct*: **fixes** *r:: 'a::len wordinterval*
shows *wordinterval-CIDR-split1 r = (Some s, u)* \implies
wordinterval-empty (wordinterval-intersection (prefix-to-wordinterval s) u)
 ⟨proof⟩ **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)
 ⟨proof⟩

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
 []
)
 ⟨proof⟩

termination *wordinterval-CIDR-split-prefixmatch*
 ⟨proof⟩ **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
 ⟨proof⟩

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

lemma *length (wordinterval-CIDR-split-prefixmatch (WordInterval 0 (0xFFFFFFFFE::32 word))) = 32* *<proof>*

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

theorem *wordinterval-CIDR-split-prefixmatch:*
wordinterval-to-set r = (∪ x ∈ set (wordinterval-CIDR-split-prefixmatch r). prefix-to-wordset x)
<proof>

lemma *wordinterval-CIDR-split-prefixmatch-all-valid-Ball: fixes r:: 'a::len wordinterval*
shows $\forall e \in \text{set (wordinterval-CIDR-split-prefixmatch r). valid-prefix } e \wedge \text{pfxm-length } e \leq \text{LENGTH('a)}$

<proof> **lemma** *wordinterval-CIDR-split-prefixmatch-all-valid-less-Ball-hlp:*
x ∈ set [s ← map (PrefixMatch x2) (pfxes TYPE('a::len0)) . valid-prefix s ∧ wordinterval-to-set (prefix-to-wordinterval s) ⊆ wordinterval-to-set rs] ⇒ pfxm-length x ≤ LENGTH('a)
<proof>

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)* *<proof>*

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>

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

<proof>

lemma *wordinterval-CIDR-split-existential:*
x ∈ wordinterval-to-set w ⇒ ∃ s. s ∈ set (wordinterval-CIDR-split-prefixmatch w) ∧ x ∈ prefix-to-wordset s
<proof>

8.1 Versions for *ipset-from-cidr*

definition *cidr-split :: 'i::len wordinterval ⇒ ('i word × nat) list* **where**
cidr-split rs ≡ map prefix-match-to-CIDR (wordinterval-CIDR-split-prefixmatch rs)

corollary *cidr-split-prefix:*

```

fixes  $r :: 'i::len$  wordinterval
shows  $(\bigcup x \in \text{set } (\text{cidr-split } r). \text{uncurry ipset-from-cidr } x) = \text{wordinterval-to-set } r$ 
   $\langle \text{proof} \rangle$ 

corollary cidr-split-prefix-single:
  fixes  $\text{start} :: 'i::len$  word
  shows  $(\bigcup x \in \text{set } (\text{cidr-split } (\text{iprange-interval } (\text{start}, \text{end}))). \text{uncurry ipset-from-cidr } x) = \{\text{start}..end\}$ 
   $\langle \text{proof} \rangle$  lemma interval-in-splitD:  $xa \in \text{foo} \implies \text{prefix-to-wordset } xa \subseteq \bigcup (\text{prefix-to-wordset } ' \text{foo})$ 
   $\langle \text{proof} \rangle$ 

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

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  $w \equiv l2wi (\text{mergesort-remdups } (wi2l w))$ 

lemma wordinterval-sort: wordinterval-to-set (wordinterval-sort  $w$ ) = wordinterval-to-set  $w$ 
   $\langle \text{proof} \rangle$ 

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

```



```

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⟩

end

```

11 Printing Machine Words

definition *string-of-word-single* :: *bool* ⇒ *'a::len word* ⇒ *string* **where**
string-of-word-single *lc w* ≡
 (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* ∘ *nat*) [*i .. j*])) :: *int* ⇒ *int* ⇒ 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"] ⟨proof⟩

function *string-of-word* :: *bool* ⇒ (*'a :: len*) *word* ⇒ *nat* ⇒ (*'a :: len*) *word* ⇒
string **where**

```

string-of-word lc base ml w =
  (if
    base < 2 ∨ LENGTH('a) < 2
  then
    undefined
  else if
    w < base ∧ 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)
  )
⟨proof⟩

```

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*

⟨proof⟩

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

lemma *hex-string-of-word0* (*0xdeadbeef42* :: *42 word*) = "*deadbeef42*" ⟨proof⟩

lemma *hex-string-of-word 1* (*0x1* :: *5 word*) = "*01*" ⟨proof⟩

lemma *hex-string-of-word 8* (*0xff*::*32 word*) = "*000000ff*" ⟨proof⟩

lemma *dec-string-of-word0* (*8*::*32 word*) = "*8*" ⟨proof⟩

lemma *dec-string-of-word0* (*3*::*2 word*) = "*11*" ⟨proof⟩

lemma *dec-string-of-word0* (*-1*::*8 word*) = "*255*" ⟨proof⟩

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

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

⟨proof⟩

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

⟨proof⟩

lemma *string-of-word-base-ten-zero-pad*:

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⟩

```

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)
  ⟨proof⟩

```

```

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

```

12 Printing Lists

```

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

```

```

definition list-separated-toString :: string ⇒ ('a ⇒ string) ⇒ 'a list ⇒ 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 ⇒ ('a ⇒ string) ⇒ 'a list ⇒ 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⟩

```

```

lemma list-separated-toString-intersperse:
  intersperse sep (map f xs) = list-separated-toString [sep] f xs
  ⟨proof⟩

```

```

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]" ⟨proof⟩

```

```

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]" <proof>

```

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"
<proof>

```

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) partslist)
)

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0xDB8 0x0 0x0 0x8 0x800 0x200C 0x417A))*
= "2001:db8::8:800:200c:417a" <proof>

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0xFF01 0x0 0x0 0x0 0x0 0x0 0x0101))* =
"ff01::101" <proof>

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

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0xDB8 0 0 0 0 0 0))* =
"2001:db8::" <proof>

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0xFF00 0 0 0 0 0 0))* =
"ff00::" <proof>

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0xFE80 0 0 0 0 0 0))* =
"fe80::" <proof>

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0 0 0 0 0 0 0 0))* =
"::" <proof>

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0 0 0 0 0 0 0 1))* =
"::1" <proof>

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0xdb8 0x0 0x0 0x0 0x0 0x0 0x1))* =
"2001:db8::1" <proof>

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0xdb8 0x0 0x0 0x0 0x0 0x2 0x1))* =
"2001:db8::2:1" <proof>

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0xdb8 0x0 0x1 0x1 0x1 0x1 0x1))* =
"2001:db8:0:1:1:1:1:1" <proof>

lemma *ipv6addr-toString (ipv6preferred-to-int (IPv6AddrPreferred 0x2001 0x0 0x0 0x1 0x0 0x0 0x0 0x1))* =

"2001:0:0:1::1" <proof>

lemma *ipv6addr-toString (ipv6preferred-to-int*
(IPv6AddrPreferred 0x2001 0xdb8 0x0 0x0 0x1 0x0 0x0 0x1)) =
"2001:db8::1:0:0:1" <proof>

lemma *ipv6addr-toString max-ipv6-addr = "fff:fff:fff:fff:fff:fff:fff:fff" <proof>*

lemma *ipv6addr-toString (ipv6preferred-to-int*
(IPv6AddrPreferred 0xffff 0xffff 0xffff 0xffff 0xffff 0xffff 0xffff 0xffff)) =
"fff:fff:fff:fff:fff:fff:fff:fff" <proof>

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 ≠ 32 then "/" @ string-of-nat l else []))

definition *prefix-match-128-toString :: 128 prefix-match ⇒ string where*
prefix-match-128-toString pfx = (case pfx of PrefixMatch p l ⇒ ipv6addr-toString
p @ (if l ≠ 128 then "/" @ string-of-nat l else []))

end