

Native Words

Andreas Lochbihler

May 26, 2024

Abstract

This entry makes machine words and machine arithmetic available for code generation from Isabelle/HOL. It provides a common abstraction that hides the differences between the different target languages. The code generator maps these operations to the APIs of the target languages. Apart from that, we extend the available bit operations on types `int` and `integer`, and map them to the operations in the target languages.

Contents

1	Common base for target language implementations of word types	5
1.0.1	More on conversions	5
1.0.2	Quickcheck conversion functions	5
1.0.3	More on division	7
1.0.4	More on misc operations	8
1.0.5	Code generator setup	9
2	Systematic approach towards type copies of word type	11
3	A special case of a conversion.	17
4	Bit operations for target language integers	19
4.1	More lemmas about <i>integers</i>	19
4.2	Target language implementations	20
4.3	Test code generator setup	31
5	Unsigned words of 64 bits	33
5.1	Type definition and primitive operations	33
5.2	Code setup	36
5.3	Quickcheck setup	48
6	Unsigned words of 32 bits	49
6.1	Type definition and primitive operations	49
6.2	Code setup	52
6.3	Quickcheck setup	62
7	Unsigned words of 16 bits	63
7.1	Type definition and primitive operations	63
7.2	Code setup	66
7.3	Quickcheck setup	73

8	Unsigned words of 8 bits	75
8.1	Type definition and primitive operations	75
8.2	Code setup	78
8.3	Quickcheck setup	86
9	Unsigned words of default size	89
9.1	Type definition and primitive operations	90
9.2	Code setup	92
9.3	Quickcheck setup	104
10	Conversions between unsigned words and between char	107
10.1	Conversion between native words	107
10.2	Compatibility with Imperative/HOL	112
11	Test cases	113
11.1	Tests for <i>uint32</i>	113
11.2	Tests for <i>uint16</i>	115
11.3	Tests for <i>uint8</i>	117
11.4	Tests for <i>uint</i>	118
11.5	Tests for <i>uint64</i>	120
11.6	Tests for casts	122
12	Implementation of bit operations on int by target language operations	125
12.1	Test cases for emulation of native words	126
12.1.1	Tests for <i>uint16</i>	126
12.1.2	Tests for <i>uint8</i>	127
12.2	Test with PolyML	127
12.3	Test with Scala	128
12.4	Test with MLton	129
13	User guide for native words	131
13.1	Lifting functions from <i>'a word</i> to native words	132
13.2	Storing native words in datatypes	133
13.2.1	Example: expressions and two semantics	133
13.2.2	Change the datatype to use machine words	134
13.2.3	Make functions use functions on machine words	135
13.3	Troubleshooting	135
13.3.1	<i>export-code</i> raises an exception	135
13.3.2	The generated code does not compile	136
13.3.3	The generated code is too slow	136

Chapter 1

Common base for target language implementations of word types

```
theory Code-Target-Word-Base
imports
  HOL-Library.Word
  Word-Lib.Signed-Division-Word
  Word-Lib.More-Word
begin
```

1.0.1 More on conversions

```
lemma int-of-integer-unsigned-eq [simp]:
  ⟨int-of-integer (unsigned w) = uint w⟩
  ⟨proof⟩
```

```
lemma int-of-integer-signed-eq [simp]:
  ⟨int-of-integer (signed w) = sint w⟩
  ⟨proof⟩
```

```
abbreviation word-of-integer :: ⟨integer ⇒ 'a::len word⟩
  where ⟨word-of-integer k ≡ word-of-int (int-of-integer k)⟩
```

1.0.2 Quickcheck conversion functions

```
context
  includes state-combinator-syntax
begin
```

```
definition qc-random-cnv ::
  (natural ⇒ 'a::term-of) ⇒ natural ⇒ Random.seed
  ⇒ ('a × (unit ⇒ Code-Evaluation.term)) × Random.seed
  where qc-random-cnv a-of-natural i = Random.range (i + 1) ◦→ (λk. Pair (
```

```

    let n = a-of-natural k
    in (n, λ-. Code-Evaluation.term-of n))

```

end

definition *qc-exhaustive-cnv* :: (natural ⇒ 'a) ⇒ ('a ⇒ (bool × term list) option)
 ⇒ natural ⇒ (bool × term list) option

where

```

    qc-exhaustive-cnv a-of-natural f d =
    Quickcheck-Exhaustive.exhaustive (%x. f (a-of-natural x)) d

```

definition *qc-full-exhaustive-cnv* ::

```

    (natural ⇒ ('a::term-of)) ⇒ ('a × (unit ⇒ term) ⇒ (bool × term list) option)
    ⇒ natural ⇒ (bool × term list) option

```

where

```

    qc-full-exhaustive-cnv a-of-natural f d = Quickcheck-Exhaustive.full-exhaustive
    (%(x, xt). f (a-of-natural x, %- Code-Evaluation.term-of (a-of-natural x))) d

```

declare [[*quickcheck-narrowing-ghc-options* = -XTypeSynonymInstances]]

definition *qc-narrowing-drawn-from* :: 'a list ⇒ integer ⇒ -

where

```

    qc-narrowing-drawn-from xs =
    foldr Quickcheck-Narrowing.sum (map Quickcheck-Narrowing.cons (butlast xs))
    (Quickcheck-Narrowing.cons (last xs))

```

locale *quickcheck-narrowing-samples* =

fixes *a-of-integer* :: integer ⇒ 'a × 'a :: {*partial-term-of*, *term-of*}

and *zero* :: 'a

and *tr* :: typerep

begin

function *narrowing-samples* :: integer ⇒ 'a list

where

```

    narrowing-samples i =

```

```

    (if i > 0 then let (a, a') = a-of-integer i in narrowing-samples (i - 1) @ [a, a']
    else [zero])

```

⟨proof⟩

termination including *integer.lifting*

⟨proof⟩

definition *partial-term-of-sample* :: integer ⇒ 'a

where

```

    partial-term-of-sample i =

```

```

    (if i < 0 then undefined

```

```

    else if i = 0 then zero

```

```

    else if i mod 2 = 0 then snd (a-of-integer (i div 2))

```

```

    else fst (a-of-integer (i div 2 + 1)))

```

```

lemma partial-term-of-code:
  partial-term-of (ty :: 'a itself) (Quickcheck-Narrowing.Narrowing-variable p t) ≡
    Code-Evaluation.Free (STR "'-' tr
  partial-term-of (ty :: 'a itself) (Quickcheck-Narrowing.Narrowing-constructor i
  []) ≡
    Code-Evaluation.term-of (partial-term-of-sample i)
  ⟨proof⟩

```

end

```

lemmas [code] =
  quickcheck-narrowing-samples.narrowing-samples.simps
  quickcheck-narrowing-samples.partial-term-of-sample-def

```

1.0.3 More on division

```

lemma div-half-nat:
  fixes x y :: nat
  assumes y ≠ 0
  shows (x div y, x mod y) = (let q = 2 * (x div 2 div y); r = x - q * y in if y ≤
  r then (q + 1, r - y) else (q, r))
  ⟨proof⟩

```

```

lemma div-half-word:
  fixes x y :: 'a :: len word
  assumes y ≠ 0
  shows (x div y, x mod y) = (let q = push-bit 1 (drop-bit 1 x div y); r = x - q
  * y in if y ≤ r then (q + 1, r - y) else (q, r))
  ⟨proof⟩

```

Division on 'a word is unsigned, but Scala and OCaml only have signed division and modulus.

```

lemma [code]:
  x sdiv y =
    (let x' = sint x; y' = sint y;
      negative = (x' < 0) ≠ (y' < 0);
      result = abs x' div abs y'
    in word-of-int (if negative then -result else result))
  for x y :: '<a::len word>
  ⟨proof⟩

```

```

lemma [code]:
  x smod y =
    (let x' = sint x; y' = sint y;
      negative = (x' < 0);
      result = abs x' mod abs y'
    in word-of-int (if negative then -result else result))
  for x y :: '<a::len word>
  ⟨proof⟩

```

This algorithm implements unsigned division in terms of signed division. Taken from Hacker's Delight.

lemma *divmod-via-sdivmod*:

fixes $x\ y :: 'a :: \text{len word}$

assumes $y \neq 0$

shows

$(x \text{ div } y, x \text{ mod } y) =$

$(\text{if push-bit } (\text{LENGTH}('a) - 1) \ 1 \leq y \text{ then if } x < y \text{ then } (0, x) \text{ else } (1, x - y)$

$\text{else let } q = (\text{push-bit } 1 \ (\text{drop-bit } 1 \ x \text{ sdiv } y));$

$r = x - q * y$

$\text{in if } r \geq y \text{ then } (q + 1, r - y) \text{ else } (q, r))$

$\langle \text{proof} \rangle$

1.0.4 More on misc operations

context

includes *bit-operations-syntax*

begin

lemma *word-of-int-code*:

$\text{uint } (\text{word-of-int } x :: 'a \text{ word}) = x \text{ AND mask } (\text{LENGTH}('a :: \text{len}))$

$\langle \text{proof} \rangle$

lemma *word-and-mask-or-conv-and-mask*:

$\text{bit } n \text{ index} \implies (n \text{ AND mask } \text{index}) \text{ OR } (\text{push-bit } \text{index } 1) = n \text{ AND mask}$

$(\text{index} + 1)$

for $n :: \langle 'a :: \text{len word} \rangle$

$\langle \text{proof} \rangle$

lemma *uint-and-mask-or-full*:

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

assumes $\text{bit } n \ (\text{LENGTH}('a) - 1)$

and $\text{mask1} = \text{mask } (\text{LENGTH}('a) - 1)$

and $\text{mask2} = \text{push-bit } (\text{LENGTH}('a) - 1) \ 1$

shows $\text{uint } (n \text{ AND mask1}) \text{ OR mask2} = \text{uint } n$

$\langle \text{proof} \rangle$

lemma *word-of-int-via-signed*:

fixes mask

assumes $\text{mask-def: } \text{mask} = \text{Bit-Operations.mask } \text{LENGTH}('a)$

and $\text{shift-def: } \text{shift} = \text{push-bit } \text{LENGTH}('a) \ 1$

and $\text{index-def: } \text{index} = \text{LENGTH}('a) - 1$

and $\text{overflow-def: } \text{overflow} = \text{push-bit } (\text{LENGTH}('a) - 1) \ 1$

and $\text{least-def: } \text{least} = - \text{overflow}$

shows

$(\text{word-of-int } i :: 'a :: \text{len word}) =$

$(\text{let } i' = i \text{ AND mask}$

$\text{in if bit } i' \ \text{index} \text{ then}$

$\text{if } i' - \text{shift} < \text{least} \vee \text{overflow} \leq i' - \text{shift} \text{ then arbitrary1 } i' \text{ else word-of-int}$


```

(i' - shift)
  else if i' < least ∨ overflow ≤ i' then arbitrary2 i' else word-of-int i')
⟨proof⟩

```

end

1.0.5 Code generator setup

The separate code target *SML-word* collects setups for the code generator that PolyML does not provide.

⟨*ML*⟩

```

code-identifier code-module Code-Target-Word-Base ↯
  (SML) Word and (Haskell) Word and (OCaml) Word and (Scala) Word

```

Misc

```

lemmas word-sdiv-def = sdiv-word-def
lemmas word-smod-def = smod-word-def

```

end

Chapter 2

Systematic approach towards type copies of word type

theory *Word-Type-Copies*

imports

HOL-Library.Word

Word-Lib.Most-significant-bit

Word-Lib.Least-significant-bit

Word-Lib.Generic-set-bit

Word-Lib.Bit-Comprehension

Code-Target-Word-Base

begin

The lifting machinery is not localized, hence the abstract proofs are carried out using morphisms.

locale *word-type-copy* =

fixes *of-word* :: $\langle 'b::\text{len } \text{word} \Rightarrow 'a \rangle$

and *word-of* :: $\langle 'a \Rightarrow 'b \text{ word} \rangle$

assumes *type-definition*: $\langle \text{type-definition } \text{word-of } \text{of-word } \text{UNIV} \rangle$

begin

lemma *word-of-word*:

$\langle \text{word-of } (\text{of-word } w) = w \rangle$

$\langle \text{proof} \rangle$

lemma *of-word-of* [*code abstype*]:

$\langle \text{of-word } (\text{word-of } p) = p \rangle$

— Use an abstract type for code generation to disable pattern matching on *of-word*.

$\langle \text{proof} \rangle$

lemma *word-of-eqI*:

$\langle p = q \rangle$ **if** $\langle \text{word-of } p = \text{word-of } q \rangle$

$\langle \text{proof} \rangle$

lemma *eq-iff-word-of*:

$\langle p = q \iff \text{word-of } p = \text{word-of } q \rangle$
 $\langle \text{proof} \rangle$

end

bundle *constraintless*

begin

$\langle ML \rangle$

end

locale *word-type-copy-ring = word-type-copy*

opening *constraintless +*

constrains *word-of* :: $\langle 'a \Rightarrow 'b::\text{len word} \rangle$

assumes *word-of-0* [code]: $\langle \text{word-of } 0 = 0 \rangle$

and *word-of-1* [code]: $\langle \text{word-of } 1 = 1 \rangle$

and *word-of-add* [code]: $\langle \text{word-of } (p + q) = \text{word-of } p + \text{word-of } q \rangle$

and *word-of-minus* [code]: $\langle \text{word-of } (-p) = -(\text{word-of } p) \rangle$

and *word-of-diff* [code]: $\langle \text{word-of } (p - q) = \text{word-of } p - \text{word-of } q \rangle$

and *word-of-mult* [code]: $\langle \text{word-of } (p * q) = \text{word-of } p * \text{word-of } q \rangle$

and *word-of-div* [code]: $\langle \text{word-of } (p \text{ div } q) = \text{word-of } p \text{ div } \text{word-of } q \rangle$

and *word-of-mod* [code]: $\langle \text{word-of } (p \text{ mod } q) = \text{word-of } p \text{ mod } \text{word-of } q \rangle$

and *equal-iff-word-of* [code]: $\langle \text{HOL.equal } p \ q \iff \text{HOL.equal } (\text{word-of } p) \ (\text{word-of } q) \rangle$

and *less-eq-iff-word-of* [code]: $\langle p \leq q \iff \text{word-of } p \leq \text{word-of } q \rangle$

and *less-iff-word-of* [code]: $\langle p < q \iff \text{word-of } p < \text{word-of } q \rangle$

begin

lemma *of-class-comm-ring-1*:

$\langle \text{OFCLASS}('a, \text{comm-ring-1-class}) \rangle$

$\langle \text{proof} \rangle$

lemma *of-class-semiring-modulo*:

$\langle \text{OFCLASS}('a, \text{semiring-modulo-class}) \rangle$

$\langle \text{proof} \rangle$

lemma *of-class-equal*:

$\langle \text{OFCLASS}('a, \text{equal-class}) \rangle$

$\langle \text{proof} \rangle$

lemma *of-class-linorder*:

$\langle \text{OFCLASS}('a, \text{linorder-class}) \rangle$

$\langle \text{proof} \rangle$

end

locale *word-type-copy-bits = word-type-copy-ring*

opening *constraintless bit-operations-syntax* +
constrains *word-of* :: $\langle 'a::\{\text{comm-ring-1, semiring-modulo, equal, linorder}\} \Rightarrow$
 $'b::\text{len word}\rangle$
fixes *signed-drop-bit* :: $\langle \text{nat} \Rightarrow 'a \Rightarrow 'a \rangle$
assumes *bit-eq-word-of* [code]: $\langle \text{bit } p = \text{bit } (\text{word-of } p) \rangle$
and *word-of-not* [code]: $\langle \text{word-of } (\text{NOT } p) = \text{NOT } (\text{word-of } p) \rangle$
and *word-of-and* [code]: $\langle \text{word-of } (p \text{ AND } q) = \text{word-of } p \text{ AND } \text{word-of } q \rangle$
and *word-of-or* [code]: $\langle \text{word-of } (p \text{ OR } q) = \text{word-of } p \text{ OR } \text{word-of } q \rangle$
and *word-of-xor* [code]: $\langle \text{word-of } (p \text{ XOR } q) = \text{word-of } p \text{ XOR } \text{word-of } q \rangle$
and *word-of-mask* [code]: $\langle \text{word-of } (\text{mask } n) = \text{mask } n \rangle$
and *word-of-push-bit* [code]: $\langle \text{word-of } (\text{push-bit } n \ p) = \text{push-bit } n \ (\text{word-of } p) \rangle$
and *word-of-drop-bit* [code]: $\langle \text{word-of } (\text{drop-bit } n \ p) = \text{drop-bit } n \ (\text{word-of } p) \rangle$
and *word-of-signed-drop-bit* [code]: $\langle \text{word-of } (\text{signed-drop-bit } n \ p) = \text{Word.signed-drop-bit}$
 $n \ (\text{word-of } p) \rangle$
and *word-of-take-bit* [code]: $\langle \text{word-of } (\text{take-bit } n \ p) = \text{take-bit } n \ (\text{word-of } p) \rangle$
and *word-of-set-bit* [code]: $\langle \text{word-of } (\text{Bit-Operations.set-bit } n \ p) = \text{Bit-Operations.set-bit}$
 $n \ (\text{word-of } p) \rangle$
and *word-of-unset-bit* [code]: $\langle \text{word-of } (\text{unset-bit } n \ p) = \text{unset-bit } n \ (\text{word-of}$
 $p) \rangle$
and *word-of-flip-bit* [code]: $\langle \text{word-of } (\text{flip-bit } n \ p) = \text{flip-bit } n \ (\text{word-of } p) \rangle$
begin

lemma *word-of-bool*:
 $\langle \text{word-of } (\text{of-bool } n) = \text{of-bool } n \rangle$
 $\langle \text{proof} \rangle$

lemma *word-of-nat*:
 $\langle \text{word-of } (\text{of-nat } n) = \text{of-nat } n \rangle$
 $\langle \text{proof} \rangle$

lemma *word-of-numeral* [simp]:
 $\langle \text{word-of } (\text{numeral } n) = \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *word-of-power*:
 $\langle \text{word-of } (p \ ^n) = \text{word-of } p \ ^n \rangle$
 $\langle \text{proof} \rangle$

lemma *even-iff-word-of*:
 $\langle 2 \ \text{dvd } p \iff 2 \ \text{dvd } \text{word-of } p \rangle$ (**is** $\langle ?P \iff ?Q \rangle$)
 $\langle \text{proof} \rangle$

lemma *of-class-ring-bit-operations*:
 $\langle \text{OFCLASS}('a, \text{ring-bit-operations-class}) \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{take-bit } n \ a = a \ \text{AND} \ \text{mask } n \rangle$ **for** $a :: 'a$
 $\langle \text{proof} \rangle$

lemma [code]:

⟨*mask* (*Suc n*) = *push-bit n* (*1 :: 'a*) *OR mask n*⟩
 ⟨*mask 0* = (*0 :: 'a*)⟩
 ⟨*proof*⟩

lemma [code]:

⟨*Bit-Operations.set-bit n w* = *w OR push-bit n 1*⟩ **for** *w :: 'a*
 ⟨*proof*⟩

lemma [code]:

⟨*unset-bit n w* = *w AND NOT (push-bit n 1)*⟩ **for** *w :: 'a*
 ⟨*proof*⟩

lemma [code]:

⟨*flip-bit n w* = *w XOR push-bit n 1*⟩ **for** *w :: 'a*
 ⟨*proof*⟩

end

locale *word-type-copy-more* = *word-type-copy-bits* +
constrains *word-of* :: ⟨*a*::{*ring-bit-operations*, *equal*, *linorder*} ⇒ *b*::*len word*⟩
fixes *of-nat* :: ⟨*nat* ⇒ *'a*⟩ **and** *nat-of* :: ⟨*'a* ⇒ *nat*⟩
and *of-int* :: ⟨*int* ⇒ *'a*⟩ **and** *int-of* :: ⟨*'a* ⇒ *int*⟩
and *of-integer* :: ⟨*integer* ⇒ *'a*⟩ **and** *integer-of* :: ⟨*'a* ⇒ *integer*⟩
assumes *word-of-nat-eq*: ⟨*word-of (of-nat n)* = *word-of-nat n*⟩
and *nat-of-eq-word-of*: ⟨*nat-of p* = *unat (word-of p)*⟩
and *word-of-int-eq*: ⟨*word-of (of-int k)* = *word-of-int k*⟩
and *int-of-eq-word-of*: ⟨*int-of p* = *uint (word-of p)*⟩
and *word-of-integer-eq*: ⟨*word-of (of-integer l)* = *word-of-integer l*⟩
and *integer-of-eq-word-of*: ⟨*integer-of p* = *unsigned (word-of p)*⟩
begin

lemma *of-word-numeral* [code-post]:

⟨*of-word (numeral n)* = *numeral n*⟩
 ⟨*proof*⟩

lemma *of-word-0* [code-post]:

⟨*of-word 0* = *0*⟩
 ⟨*proof*⟩

lemma *of-word-1* [code-post]:

⟨*of-word 1* = *1*⟩
 ⟨*proof*⟩

Use pretty numerals from integer for pretty printing

lemma *numeral-eq-integer* [code-unfold]:

⟨*numeral n* = *of-integer (numeral n)*⟩
 ⟨*proof*⟩

```

lemma neg-numeral-eq-integer [code-unfold]:
  ⟨− numeral n = of-integer (− numeral n)⟩
  ⟨proof⟩

end

locale word-type-copy-misc = word-type-copy-more
  opening constraintless bit-operations-syntax +
  constrains word-of :: ⟨'a::{ring-bit-operations, equal, linorder} ⇒ 'b::len word⟩
  fixes size :: nat and set-bits-aux :: ⟨(nat ⇒ bool) ⇒ nat ⇒ 'a ⇒ 'a⟩
  assumes size-eq-length: ⟨size = LENGTH('b::len)⟩
  and msb-iff-word-of [code]: ⟨msb p ⟷ msb (word-of p)⟩
  and lsb-iff-word-of [code]: ⟨lsb p ⟷ lsb (word-of p)⟩
  and size-eq-word-of: ⟨Nat.size (p :: 'a) = Nat.size (word-of p)⟩
  and word-of-generic-set-bit [code]: ⟨word-of (Generic-set-bit.set-bit p n b) =
Generic-set-bit.set-bit (word-of p) n b⟩
  and word-of-set-bits: ⟨word-of (set-bits P) = set-bits P⟩
  and word-of-set-bits-aux: ⟨word-of (set-bits-aux P n p) = Bit-Comprehension.set-bits-aux
P n (word-of p)⟩
begin

lemma size-eq [code]:
  ⟨Nat.size p = size⟩ for p :: 'a
  ⟨proof⟩

lemma set-bits-aux-code [code]:
  ⟨set-bits-aux f n w =
  (if n = 0 then w
  else let n' = n − 1 in set-bits-aux f n' (push-bit 1 w OR (if f n' then 1 else 0)))⟩
  ⟨proof⟩

lemma set-bits-code [code]: ⟨set-bits P = set-bits-aux P size 0⟩
  ⟨proof⟩

lemma of-class-lsb:
  ⟨OFCLASS('a, lsb-class)⟩
  ⟨proof⟩

lemma of-class-set-bit:
  ⟨OFCLASS('a, set-bit-class)⟩
  ⟨proof⟩

lemma of-class-bit-comprehension:
  ⟨OFCLASS('a, bit-comprehension-class)⟩
  ⟨proof⟩

end

```

end

Chapter 3

A special case of a conversion.

```
theory Code-Int-Integer-Conversion
imports
  Main
begin
```

Use this function to convert numeral *integers* quickly into *ints*. By default, it works only for symbolic evaluation; normally generated code raises an exception at run-time. If theory *Code-Target-Int-Bit* is imported, it works again, because then *int* is implemented in terms of *integer* even for symbolic evaluation.

```
definition int-of-integer-symbolic :: integer  $\Rightarrow$  int
  where int-of-integer-symbolic = int-of-integer
```

```
lemma int-of-integer-symbolic-aux-code [code nbe]:
  int-of-integer-symbolic 0 = 0
  int-of-integer-symbolic (Code-Numeral.Pos n) = Int.Pos n
  int-of-integer-symbolic (Code-Numeral.Neg n) = Int.Neg n
  <proof>
```

```
end
```


Chapter 4

Bit operations for target language integers

```
theory Code-Target-Integer-Bit
imports
  HOL-Library.Word
  Code-Int-Integer-Conversion
  Word-Lib.Most-significant-bit
  Word-Lib.Least-significant-bit
  Word-Lib.Generic-set-bit
  Word-Lib.Bit-Comprehension
begin

TODO: separate

lemmas [transfer-rule] =
  identity-quotient
  fun-quotient
  Quotient-integer[folded integer.pcr-cr-eq]

lemma undefined-transfer:
  assumes Quotient R Abs Rep T
  shows T (Rep undefined) undefined
  ⟨proof⟩

bundle undefined-transfer = undefined-transfer[transfer-rule]
```

4.1 More lemmas about *integers*

```
context
includes integer.lifting
begin

lemma integer-of-nat-less-0-conv [simp]:  $\neg$  integer-of-nat  $n < 0$ 
  ⟨proof⟩
```

lemma *int-of-integer-pow*: $\text{int-of-integer } (x \wedge n) = \text{int-of-integer } x \wedge n$
 ⟨proof⟩

lemma *sub1-lt-0-iff* [simp]: $\text{Code-N numeral.sub } n \text{ num.One} < 0 \longleftrightarrow \text{False}$
 ⟨proof⟩

lemma *nat-of-integer-numeral* [simp]: $\text{nat-of-integer } (\text{numeral } n) = \text{numeral } n$
 ⟨proof⟩

lemma *nat-of-integer-sub1-conv-pred-numeral* [simp]:
 $\text{nat-of-integer } (\text{Code-N numeral.sub } n \text{ num.One}) = \text{pred-numeral } n$
 ⟨proof⟩

lemma *nat-of-integer-1* [simp]: $\text{nat-of-integer } 1 = 1$
 ⟨proof⟩

lemma *dup-1* [simp]: $\text{Code-N numeral.dup } 1 = 2$
 ⟨proof⟩

lift-definition *bin-rest-integer* :: $\text{integer} \Rightarrow \text{integer}$ **is** $\langle \lambda k . k \text{ div } 2 \rangle$ ⟨proof⟩

lift-definition *bin-last-integer* :: $\text{integer} \Rightarrow \text{bool}$ **is** *odd* ⟨proof⟩

lift-definition *Bit-integer* :: $\text{integer} \Rightarrow \text{bool} \Rightarrow \text{integer}$ **is** $\langle \lambda k b . \text{of-bool } b + 2 * k \rangle$
 ⟨proof⟩

end

4.2 Target language implementations

Unfortunately, this is not straightforward, because these API functions have different signatures and preconditions on the parameters:

Standard ML Shifts in `IntInf` are given as `word`, but not `IntInf`.

Haskell In the `Data.Bits.Bits` type class, shifts and bit indices are given as `Int` rather than `Integer`.

Additional constants take only parameters of type *integer* rather than *nat* and check the preconditions as far as possible (e.g., being non-negative) in a portable way. Manual implementations inside `code_printing` perform the remaining range checks and convert these *integers* into the right type.

For normalisation by evaluation, we derive custom code equations, because NBE does not know these `code_printing` serialisations and would otherwise loop.

code-printing code-module *Integer-Bit* \rightarrow (SML)

```

⟨structure Integer-Bit : sig
  val test-bit : IntInf.int -> IntInf.int -> bool
  val set-bit : IntInf.int -> IntInf.int -> bool -> IntInf.int
  val shiftl : IntInf.int -> IntInf.int -> IntInf.int
  val shiftr : IntInf.int -> IntInf.int -> IntInf.int
end = struct

val maxWord = IntInf.pow (2, Word.wordSize);

fun test-bit x n =
  if n < maxWord then IntInf.andb (x, IntInf.<<< (1, Word.fromLargeInt (IntInf.toLarge
n))) <> 0
  else raise (Fail (Bit index too large: ^ IntInf.toString n));

fun set-bit x n b =
  if n < maxWord then
    if b then IntInf.orb (x, IntInf.<<< (1, Word.fromLargeInt (IntInf.toLarge n)))
    else IntInf.andb (x, IntInf.notb (IntInf.<<< (1, Word.fromLargeInt (IntInf.toLarge
n))))
  else raise (Fail (Bit index too large: ^ IntInf.toString n));

fun shiftl x n =
  if n < maxWord then IntInf.<<< (x, Word.fromLargeInt (IntInf.toLarge n))
  else raise (Fail (Shift operand too large: ^ IntInf.toString n));

fun shiftr x n =
  if n < maxWord then IntInf.~>> (x, Word.fromLargeInt (IntInf.toLarge n))
  else raise (Fail (Shift operand too large: ^ IntInf.toString n));

end; (*struct Integer-Bit*)
code-reserved SML Integer-Bit

code-printing code-module Integer-Bit ↪ (OCaml)
⟨module Integer-Bit : sig
  val test-bit : Z.t -> Z.t -> bool
  val shiftl : Z.t -> Z.t -> Z.t
  val shiftr : Z.t -> Z.t -> Z.t
end = struct

(* We do not need an explicit range checks here,
  because Big-int.int-of-big-int raises Failure
  if the argument does not fit into an int. *)
let test-bit x n = Z.testbit x (Z.to-int n);;

let shiftl x n = Z.shift-left x (Z.to-int n);;

let shiftr x n = Z.shift-right x (Z.to-int n);;

end;; (*struct Integer-Bit*)

```

22 CHAPTER 4. BIT OPERATIONS FOR TARGET LANGUAGE INTEGERS

```

code-reserved OCaml Integer-Bit

code-printing code-module Data-Bits  $\rightarrow$  (Haskell)
<
module Data-Bits where {

import qualified Data.Bits;

{-
  The ...Bounded functions assume that the Integer argument for the shift
  or bit index fits into an Int, is non-negative and (for types of fixed bit width)
  less than bitSize
-}

infixl 7 .&;
infixl 6 'xor';
infixl 5 .|;

(.&.) :: Data.Bits.Bits a => a -> a -> a;
(.&.) = (Data.Bits.&.);

xor :: Data.Bits.Bits a => a -> a -> a;
xor = Data.Bits.xor;

(.|.) :: Data.Bits.Bits a => a -> a -> a;
(.|.) = (Data.Bits.|.);

complement :: Data.Bits.Bits a => a -> a;
complement = Data.Bits.complement;

testBitUnbounded :: Data.Bits.Bits a => a -> Integer -> Bool;
testBitUnbounded x b
  | b <= toInteger (Prelude.maxBound :: Int) = Data.Bits.testBit x (fromInteger
b)
  | otherwise = error (Bit index too large: ++ show b)
;

testBitBounded :: Data.Bits.Bits a => a -> Integer -> Bool;
testBitBounded x b = Data.Bits.testBit x (fromInteger b);

setBitUnbounded :: Data.Bits.Bits a => a -> Integer -> Bool -> a;
setBitUnbounded x n b
  | n <= toInteger (Prelude.maxBound :: Int) =
    if b then Data.Bits.setBit x (fromInteger n) else Data.Bits.clearBit x (fromInteger
n)
  | otherwise = error (Bit index too large: ++ show n)
;

setBitBounded :: Data.Bits.Bits a => a -> Integer -> Bool -> a;

```

```

setBitBounded x n True = Data.Bits.setBit x (fromInteger n);
setBitBounded x n False = Data.Bits.clearBit x (fromInteger n);

shifftlUnbounded :: Data.Bits.Bits a => a -> Integer -> a;
shifftlUnbounded x n
  | n <= toInteger (Prelude.maxBound :: Int) = Data.Bits.shiftL x (fromInteger
n)
  | otherwise = error (Shift operand too large: ++ show n)
;

shifftlBounded :: Data.Bits.Bits a => a -> Integer -> a;
shifftlBounded x n = Data.Bits.shiftL x (fromInteger n);

shiftrUnbounded :: Data.Bits.Bits a => a -> Integer -> a;
shiftrUnbounded x n
  | n <= toInteger (Prelude.maxBound :: Int) = Data.Bits.shiftR x (fromInteger
n)
  | otherwise = error (Shift operand too large: ++ show n)
;

shiftrBounded :: (Ord a, Data.Bits.Bits a) => a -> Integer -> a;
shiftrBounded x n = Data.Bits.shiftR x (fromInteger n);

}

```

and — *HOL.Quickcheck-Narrowing* maps *integer* to Haskell’s `Prelude.Int` type instead of `Integer`. For compatibility with the Haskell target, we nevertheless provide bounded and unbounded functions.

(*Haskell-Quickcheck*)

```

<
module Data-Bits where {

import qualified Data.Bits;

{–
  The functions assume that the Int argument for the shift or bit index is
  non-negative and (for types of fixed bit width) less than bitSize
–}

infixl 7 .&;
infixl 6 ‘xor‘;
infixl 5 .|.;

(&.) :: Data.Bits.Bits a => a -> a -> a;
(&.) = (Data.Bits.&.);

xor :: Data.Bits.Bits a => a -> a -> a;
xor = Data.Bits.xor;

```

24 CHAPTER 4. BIT OPERATIONS FOR TARGET LANGUAGE INTEGERS

```
(.|.) :: Data.Bits.Bits a => a -> a -> a;
(|.) = (Data.Bits.|.);

complement :: Data.Bits.Bits a => a -> a;
complement = Data.Bits.complement;

testBitUnbounded :: Data.Bits.Bits a => a -> Prelude.Int -> Bool;
testBitUnbounded = Data.Bits.testBit;

testBitBounded :: Data.Bits.Bits a => a -> Prelude.Int -> Bool;
testBitBounded = Data.Bits.testBit;

setBitUnbounded :: Data.Bits.Bits a => a -> Prelude.Int -> Bool -> a;
setBitUnbounded x n True = Data.Bits.setBit x n;
setBitUnbounded x n False = Data.Bits.clearBit x n;

setBitBounded :: Data.Bits.Bits a => a -> Prelude.Int -> Bool -> a;
setBitBounded x n True = Data.Bits.setBit x n;
setBitBounded x n False = Data.Bits.clearBit x n;

shiftrlUnbounded :: Data.Bits.Bits a => a -> Prelude.Int -> a;
shiftrlUnbounded = Data.Bits.shiftL;

shiftrlBounded :: Data.Bits.Bits a => a -> Prelude.Int -> a;
shiftrlBounded = Data.Bits.shiftL;

shiftrUnbounded :: Data.Bits.Bits a => a -> Prelude.Int -> a;
shiftrUnbounded = Data.Bits.shiftR;

shiftrBounded :: (Ord a, Data.Bits.Bits a) => a -> Prelude.Int -> a;
shiftrBounded = Data.Bits.shiftR;

}>
code-reserved Haskell Data-Bits
```

code-printing code-module Integer-Bit \rightarrow (Scala)

```
<object Integer-Bit {

def testBit(x: BigInt, n: BigInt) : Boolean =
  n.isValidInt match {
    case true => x.testBit(n.toInt)
    case false => sys.error("Bit index too large: + n.toString")
  }

def setBit(x: BigInt, n: BigInt, b: Boolean) : BigInt =
  n.isValidInt match {
    case true if b => x.setBit(n.toInt)
    case true => x.clearBit(n.toInt)
    case false => sys.error("Bit index too large: + n.toString")
  }
}
```



```

}

def shiftl(x: BigInt, n: BigInt) : BigInt =
  n.isValidInt match {
    case true => x << n.toInt
    case false => sys.error(Shift index too large: + n.toString)
  }

def shiftr(x: BigInt, n: BigInt) : BigInt =
  n.isValidInt match {
    case true => x << n.toInt
    case false => sys.error(Shift index too large: + n.toString)
  }

} /* object Integer-Bit */

code-printing
constant Bit-Operations.and :: integer => integer => integer →
  (SML) IntInf.andb ((-),/ (-)) and
  (OCaml) Z.logand and
  (Haskell) ((Data'-Bits.&.) :: Integer -> Integer -> Integer) and
  (Haskell-Quickcheck) ((Data'-Bits.&.) :: Prelude.Int -> Prelude.Int -> Prelude.Int) and
  (Scala) infixl 3 &
| constant Bit-Operations.or :: integer => integer => integer →
  (SML) IntInf.orb ((-),/ (-)) and
  (OCaml) Z.logor and
  (Haskell) ((Data'-Bits.|.) :: Integer -> Integer -> Integer) and
  (Haskell-Quickcheck) ((Data'-Bits.|.) :: Prelude.Int -> Prelude.Int -> Prelude.Int) and
  (Scala) infixl 1 |
| constant Bit-Operations.xor :: integer => integer => integer →
  (SML) IntInf.xorb ((-),/ (-)) and
  (OCaml) Z.logxor and
  (Haskell) (Data'-Bits.xor :: Integer -> Integer -> Integer) and
  (Haskell-Quickcheck) (Data'-Bits.xor :: Prelude.Int -> Prelude.Int -> Prelude.Int) and
  (Scala) infixl 2 ^
| constant Bit-Operations.not :: integer => integer →
  (SML) IntInf.notb and
  (OCaml) Z.lognot and
  (Haskell) (Data'-Bits.complement :: Integer -> Integer) and
  (Haskell-Quickcheck) (Data'-Bits.complement :: Prelude.Int -> Prelude.Int) and
  (Scala) -.unary'~

code-printing constant bin-rest-integer →
  (SML) IntInf.div ((-), 2) and
  (OCaml) Z.shift'-right/ -/ 1 and
  (Haskell) (Data'-Bits.shiftrUnbounded - 1 :: Integer) and

```

(*Haskell-Quickcheck*) (*Data'-Bits.shiftrUnbounded - 1 :: Prelude.Int*) and
 (*Scala*) - >> 1

context

includes *integer.lifting bit-operations-syntax*

begin

lemma *bitNOT-integer-code* [*code*]:

fixes *i :: integer* **shows**

$NOT\ i = -\ i - 1$

<proof>

lemma *bin-rest-integer-code* [*code nbe*]:

$bin\ rest\ integer\ i = i\ div\ 2$

<proof>

lemma *bin-last-integer-code* [*code*]:

$bin\ last\ integer\ i \longleftrightarrow i\ AND\ 1 \neq 0$

<proof>

lemma *bin-last-integer-nbe* [*code nbe*]:

$bin\ last\ integer\ i \longleftrightarrow i\ mod\ 2 \neq 0$

<proof>

lemma *bitval-bin-last-integer* [*code-unfold*]:

$of\ bool\ (bin\ last\ integer\ i) = i\ AND\ 1$

<proof>

end

definition *integer-test-bit* :: *integer* \Rightarrow *integer* \Rightarrow *bool*

where *integer-test-bit* *x n* = (*if* *n* < 0 *then* *undefined* *x n* *else* *bit* *x* (*nat-of-integer* *n*))

declare [[*code drop: <bit :: integer* \Rightarrow *nat* \Rightarrow *bool*]]

lemma *bit-integer-code* [*code*]:

$bit\ x\ n \longleftrightarrow integer\ test\ bit\ x\ (integer\ of\ nat\ n)$

<proof>

lemma *integer-test-bit-code* [*code*]:

$integer\ test\ bit\ x\ (Code\ Numeral.Neg\ n) = undefined\ x\ (Code\ Numeral.Neg\ n)$

$integer\ test\ bit\ 0\ 0 = False$

$integer\ test\ bit\ 0\ (Code\ Numeral.Pos\ n) = False$

$integer\ test\ bit\ (Code\ Numeral.Pos\ num.One)\ 0 = True$

$integer\ test\ bit\ (Code\ Numeral.Pos\ (num.Bit0\ n))\ 0 = False$

$integer\ test\ bit\ (Code\ Numeral.Pos\ (num.Bit1\ n))\ 0 = True$

$integer\ test\ bit\ (Code\ Numeral.Pos\ num.One)\ (Code\ Numeral.Pos\ n') = False$

$integer\ test\ bit\ (Code\ Numeral.Pos\ (num.Bit0\ n))\ (Code\ Numeral.Pos\ n') =$

```

integer-test-bit (Code-Numeral.Pos n) (Code-Numeral.sub n' num.One)
integer-test-bit (Code-Numeral.Pos (num.Bit1 n)) (Code-Numeral.Pos n') =
integer-test-bit (Code-Numeral.Pos n) (Code-Numeral.sub n' num.One)
integer-test-bit (Code-Numeral.Neg num.One) 0 = True
integer-test-bit (Code-Numeral.Neg (num.Bit0 n)) 0 = False
integer-test-bit (Code-Numeral.Neg (num.Bit1 n)) 0 = True
integer-test-bit (Code-Numeral.Neg num.One) (Code-Numeral.Pos n') = True
integer-test-bit (Code-Numeral.Neg (num.Bit0 n)) (Code-Numeral.Pos n') =
integer-test-bit (Code-Numeral.Neg n) (Code-Numeral.sub n' num.One)
integer-test-bit (Code-Numeral.Neg (num.Bit1 n)) (Code-Numeral.Pos n') =
integer-test-bit (Code-Numeral.Neg (n + num.One)) (Code-Numeral.sub n' num.One)
⟨proof⟩

```

code-printing constant *integer-test-bit* \mapsto
(SML) *Integer'-Bit.test'-bit* **and**
(OCaml) *Integer'-Bit.test'-bit* **and**
(Haskell) (*Data'-Bits.testBitUnbounded* :: *Integer* \rightarrow *Integer* \rightarrow *Bool*) **and**
(Haskell-Quickcheck) (*Data'-Bits.testBitUnbounded* :: *Prelude.Int* \rightarrow *Prelude.Int*
 \rightarrow *Bool*) **and**
(Scala) *Integer'-Bit.testBit*

context

includes *integer.lifting bit-operations-syntax*

begin

lemma *lsb-integer-code* [code]:

fixes $x :: \textit{integer}$ **shows**

$\textit{lsb } x = \textit{bit } x 0$

⟨proof⟩

definition *integer-set-bit* :: *integer* \Rightarrow *integer* \Rightarrow *bool* \Rightarrow *integer*

where [code del]: *integer-set-bit* $x n b = (\textit{if } n < 0 \textit{ then undefined } x n b \textit{ else set-bit } x (\textit{nat-of-integer } n) b)$

lemma *set-bit-integer-code* [code]:

$\textit{set-bit } x i b = \textit{integer-set-bit } x (\textit{integer-of-nat } i) b$

⟨proof⟩

lemma *set-bit-integer-conv-masks*:

fixes $x :: \textit{integer}$ **shows**

$\textit{set-bit } x i b = (\textit{if } b \textit{ then } x \textit{ OR } (\textit{push-bit } i 1) \textit{ else } x \textit{ AND NOT } (\textit{push-bit } i 1))$

⟨proof⟩

end

code-printing constant *integer-set-bit* \mapsto

(SML) *Integer'-Bit.set'-bit* **and**

(Haskell) (*Data'-Bits.setBitUnbounded* :: *Integer* \rightarrow *Integer* \rightarrow *Bool* \rightarrow *Integer*) **and**

(Haskell-Quickcheck) (*Data'-Bits.setBitUnbounded* :: *Prelude.Int* -> *Prelude.Int*
 -> *Bool* -> *Prelude.Int*) **and**
 (Scala) *Integer'-Bit.setBit*

OCaml.*Big_int* does not have a method for changing an individual bit, so we emulate that with masks. We prefer an Isabelle implementation, because this then takes care of the signs for AND and OR.

context

includes *bit-operations-syntax*

begin

lemma *integer-set-bit-code* [code]:

integer-set-bit *x n b* =
 (if *n* < 0 then undefined *x n b* else
 if *b* then *x* OR (*push-bit* (*nat-of-integer* *n*) 1)
 else *x* AND NOT (*push-bit* (*nat-of-integer* *n*) 1))
 ⟨proof⟩

end

definition *integer-shiffl* :: *integer* ⇒ *integer* ⇒ *integer*

where [code del]: *integer-shiffl* *x n* = (if *n* < 0 then undefined *x n* else *push-bit*
 (*nat-of-integer* *n*) *x*)

declare [[code drop: ⟨*push-bit* :: *nat* ⇒ *integer* ⇒ *integer*⟩]]

lemma *shiffl-integer-code* [code]:

fixes *x* :: *integer* **shows**
push-bit *n x* = *integer-shiffl* *x* (*integer-of-nat* *n*)
 ⟨proof⟩

context

includes *integer.lifting*

begin

lemma *shiffl-integer-conv-mult-pow2*:

fixes *x* :: *integer* **shows**
push-bit *n x* = *x* * 2 ^ *n*
 ⟨proof⟩

lemma *integer-shiffl-code* [code]:

integer-shiffl *x* (*Code-Numeral.Neg* *n*) = undefined *x* (*Code-Numeral.Neg* *n*)
integer-shiffl *x* 0 = *x*
integer-shiffl *x* (*Code-Numeral.Pos* *n*) = *integer-shiffl* (*Code-Numeral.dup* *x*)
 (*Code-Numeral.sub* *n* *num.One*)
integer-shiffl 0 (*Code-Numeral.Pos* *n*) = 0
 ⟨proof⟩

end

code-printing constant *integer-shiffl* \rightarrow
 (SML) *Integer'-Bit.shiffl* **and**
 (OCaml) *Integer'-Bit.shiffl* **and**
 (Haskell) (*Data'-Bits.shifflUnbounded* :: *Integer* \rightarrow *Integer* \rightarrow *Integer*) **and**
 (Haskell-Quickcheck) (*Data'-Bits.shifflUnbounded* :: *Prelude.Int* \rightarrow *Prelude.Int*
 \rightarrow *Prelude.Int*) **and**
 (Scala) *Integer'-Bit.shiffl*

definition *integer-shiftr* :: *integer* \Rightarrow *integer* \Rightarrow *integer*
where [code del]: *integer-shiftr* *x n* = (if *n* < 0 then undefined *x n* else drop-bit
 (nat-of-integer *n*) *x*)

declare [[code drop: <drop-bit :: nat \Rightarrow integer \Rightarrow integer>]]

lemma *shiftr-integer-conv-div-pow2*:
includes *integer.lifting* **fixes** *x* :: *integer* **shows**
drop-bit n x = *x div 2 ^ n*
 <proof>

lemma *shiftr-integer-code* [code]:
fixes *x* :: *integer* **shows**
drop-bit n x = *integer-shiftr x (integer-of-nat n)*
 <proof>

code-printing constant *integer-shiftr* \rightarrow
 (SML) *Integer'-Bit.shiftr* **and**
 (OCaml) *Integer'-Bit.shiftr* **and**
 (Haskell) (*Data'-Bits.shiftrUnbounded* :: *Integer* \rightarrow *Integer* \rightarrow *Integer*) **and**
 (Haskell-Quickcheck) (*Data'-Bits.shiftrUnbounded* :: *Prelude.Int* \rightarrow *Prelude.Int*
 \rightarrow *Prelude.Int*) **and**
 (Scala) *Integer'-Bit.shiftr*

lemma *integer-shiftr-code* [code]:
includes *integer.lifting*
shows
integer-shiftr x (Code-Numeral.Neg n) = undefined *x (Code-Numeral.Neg n)*
integer-shiftr x 0 = *x*
integer-shiftr 0 (Code-Numeral.Pos n) = 0
integer-shiftr (Code-Numeral.Pos num.One) (Code-Numeral.Pos n) = 0
integer-shiftr (Code-Numeral.Pos (num.Bit0 n')) (Code-Numeral.Pos n) =
integer-shiftr (Code-Numeral.Pos n') (Code-Numeral.sub n num.One)
integer-shiftr (Code-Numeral.Pos (num.Bit1 n')) (Code-Numeral.Pos n) =
integer-shiftr (Code-Numeral.Pos n') (Code-Numeral.sub n num.One)
integer-shiftr (Code-Numeral.Neg num.One) (Code-Numeral.Pos n) = -1
integer-shiftr (Code-Numeral.Neg (num.Bit0 n')) (Code-Numeral.Pos n) =
integer-shiftr (Code-Numeral.Neg n') (Code-Numeral.sub n num.One)
integer-shiftr (Code-Numeral.Neg (num.Bit1 n')) (Code-Numeral.Pos n) =
integer-shiftr (Code-Numeral.Neg (Num.inc n')) (Code-Numeral.sub n num.One)

<proof>

context
includes *integer.lifting*
begin

lemma *Bit-integer-code* [code]:
Bit-integer i False = push-bit 1 i
Bit-integer i True = push-bit 1 i + 1
<proof>

lemma *msb-integer-code* [code]:
msb (x :: integer) \longleftrightarrow x < 0
<proof>

end

context
includes *integer.lifting bit-operations-syntax*
begin

lemma *bitAND-integer-unfold* [code]:
x AND y =
(if x = 0 then 0
else if x = - 1 then y
else Bit-integer (bin-rest-integer x AND bin-rest-integer y) (bin-last-integer x \wedge
bin-last-integer y))
<proof>

lemma *bitOR-integer-unfold* [code]:
x OR y =
(if x = 0 then y
else if x = - 1 then - 1
else Bit-integer (bin-rest-integer x OR bin-rest-integer y) (bin-last-integer x \vee
bin-last-integer y))
<proof>

lemma *bitXOR-integer-unfold* [code]:
x XOR y =
(if x = 0 then y
else if x = - 1 then NOT y
else Bit-integer (bin-rest-integer x XOR bin-rest-integer y)
(\neg bin-last-integer x \longleftrightarrow bin-last-integer y))
<proof>

end

4.3 Test code generator setup

context

includes *bit-operations-syntax*

begin

definition *bit-integer-test* :: *bool* **where**

```

bit-integer-test =
  (([ -1 AND 3, 1 AND -3, 3 AND 5, -3 AND (- 5)
    , -3 OR 1, 1 OR -3, 3 OR 5, -3 OR (- 5)
    , NOT 1, NOT (- 3)
    , -1 XOR 3, 1 XOR (- 3), 3 XOR 5, -5 XOR (- 3)
    , set-bit 5 4 True, set-bit (- 5) 2 True, set-bit 5 0 False, set-bit (- 5) 1 False
    , push-bit 2 1, push-bit 3 (- 1)
    , drop-bit 3 100, drop-bit 3 (- 100)] :: integer list)
  = [ 3, 1, 1, -7
    , -3, -3, 7, -1
    , -2, 2
    , -4, -4, 6, 6
    , 21, -1, 4, -7
    , 4, -8
    , 12, -13] ^
    [ bit (5 :: integer) 4, bit (5 :: integer) 2, bit (-5 :: integer) 4, bit (-5 ::
integer) 2
    , lsb (5 :: integer), lsb (4 :: integer), lsb (-1 :: integer), lsb (-2 :: integer),
    msb (5 :: integer), msb (0 :: integer), msb (-1 :: integer), msb (-2 :: integer)]
  = [ False, True, True, False,
    True, False, True, False,
    False, False, True, True]

```

export-code *bit-integer-test* **checking** *SML Haskell? Haskell-Quickcheck? OCaml? Scala*

notepad begin

<proof>

end

<ML>

lemma $x \text{ AND } y = x \text{ OR } (y :: \text{integer})$

quickcheck[*random, expect=counterexample*]

quickcheck[*exhaustive, expect=counterexample*]

<proof>

lemma $(x :: \text{integer}) \text{ AND } x = x \text{ OR } x$

quickcheck[*narrowing, expect=no-counterexample*]

<proof>

lemma $(f :: \text{integer} \Rightarrow \text{unit}) = g$

quickcheck[*narrowing, size=3, expect=no-counterexample*]

<proof>

end

hide-const *bit-integer-test*

hide-fact *bit-integer-test-def*

end

Chapter 5

Unsigned words of 64 bits

```
theory Uint64 imports  
  Word-Type-Copies  
  Code-Target-Integer-Bit  
begin
```

PolyML (in version 5.7) provides a `Word64` structure only when run in 64-bit mode. Therefore, we by default provide an implementation of 64-bit words using `IntInf.int` and masking. The code target `SML_word` replaces this implementation and maps the operations directly to the `Word64` structure provided by the Standard ML implementations.

The `Eval` target used by `value` and `eval` dynamically tests at runtime for the version of PolyML and uses PolyML's `Word64` structure if it detects a 64-bit version which does not suffer from a division bug found in PolyML 5.6.

5.1 Type definition and primitive operations

```
typedef uint64 =  $\langle UNIV :: 64 \text{ word set} \rangle$   $\langle proof \rangle$   
  
global-interpretation uint64: word-type-copy Abs-uint64 Rep-uint64  
   $\langle proof \rangle$   
  
setup-lifting type-definition-uint64  
  
declare uint64.of-word-of [code abstype]  
  
declare Quotient-uint64 [transfer-rule]  
  
instantiation uint64 ::  $\langle \{ comm\text{-}ring\text{-}1, semiring\text{-}modulo, equal, linorder \} \rangle$   
begin  
  
lift-definition zero-uint64 :: uint64 is 0  $\langle proof \rangle$   
lift-definition one-uint64 :: uint64 is 1  $\langle proof \rangle$ 
```

lift-definition *plus-uint64* :: $\langle \text{uint64} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** $\langle (+) \rangle$ $\langle \text{proof} \rangle$
lift-definition *uminus-uint64* :: $\langle \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** *uminus* $\langle \text{proof} \rangle$
lift-definition *minus-uint64* :: $\langle \text{uint64} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** $\langle (-) \rangle$ $\langle \text{proof} \rangle$
lift-definition *times-uint64* :: $\langle \text{uint64} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** $\langle (*) \rangle$ $\langle \text{proof} \rangle$
lift-definition *divide-uint64* :: $\langle \text{uint64} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** $\langle (\text{div}) \rangle$ $\langle \text{proof} \rangle$
lift-definition *modulo-uint64* :: $\langle \text{uint64} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** $\langle (\text{mod}) \rangle$ $\langle \text{proof} \rangle$
lift-definition *equal-uint64* :: $\langle \text{uint64} \Rightarrow \text{uint64} \Rightarrow \text{bool} \rangle$ **is** $\langle \text{HOL.equal} \rangle$ $\langle \text{proof} \rangle$
lift-definition *less-eq-uint64* :: $\langle \text{uint64} \Rightarrow \text{uint64} \Rightarrow \text{bool} \rangle$ **is** $\langle (\leq) \rangle$ $\langle \text{proof} \rangle$
lift-definition *less-uint64* :: $\langle \text{uint64} \Rightarrow \text{uint64} \Rightarrow \text{bool} \rangle$ **is** $\langle (<) \rangle$ $\langle \text{proof} \rangle$

global-interpretation *uint64*: *word-type-copy-ring Abs-uint64 Rep-uint64*
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

instantiation *uint64* :: *ring-bit-operations*
begin

lift-definition *bit-uint64* :: $\langle \text{uint64} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$ **is** *bit* $\langle \text{proof} \rangle$
lift-definition *not-uint64* :: $\langle \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** $\langle \text{Bit-Operations.not} \rangle$ $\langle \text{proof} \rangle$
lift-definition *and-uint64* :: $\langle \text{uint64} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** $\langle \text{Bit-Operations.and} \rangle$
 $\langle \text{proof} \rangle$
lift-definition *or-uint64* :: $\langle \text{uint64} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** $\langle \text{Bit-Operations.or} \rangle$
 $\langle \text{proof} \rangle$
lift-definition *xor-uint64* :: $\langle \text{uint64} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** $\langle \text{Bit-Operations.xor} \rangle$
 $\langle \text{proof} \rangle$
lift-definition *mask-uint64* :: $\langle \text{nat} \Rightarrow \text{uint64} \rangle$ **is** *mask* $\langle \text{proof} \rangle$
lift-definition *push-bit-uint64* :: $\langle \text{nat} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** *push-bit* $\langle \text{proof} \rangle$
lift-definition *drop-bit-uint64* :: $\langle \text{nat} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** *drop-bit* $\langle \text{proof} \rangle$
lift-definition *signed-drop-bit-uint64* :: $\langle \text{nat} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** *signed-drop-bit*
 $\langle \text{proof} \rangle$
lift-definition *take-bit-uint64* :: $\langle \text{nat} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** *take-bit* $\langle \text{proof} \rangle$
lift-definition *set-bit-uint64* :: $\langle \text{nat} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** $\langle \text{Bit-Operations.set-bit} \rangle$
 $\langle \text{proof} \rangle$
lift-definition *unset-bit-uint64* :: $\langle \text{nat} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** *unset-bit* $\langle \text{proof} \rangle$
lift-definition *flip-bit-uint64* :: $\langle \text{nat} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is** *flip-bit* $\langle \text{proof} \rangle$

global-interpretation *uint64*: *word-type-copy-bits Abs-uint64 Rep-uint64 signed-drop-bit-uint64*
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

lift-definition *uint64-of-nat* :: $\langle \text{nat} \Rightarrow \text{uint64} \rangle$
is *word-of-nat* $\langle \text{proof} \rangle$

lift-definition *nat-of-uint64* :: $\langle \text{uint64} \Rightarrow \text{nat} \rangle$
is *unat* $\langle \text{proof} \rangle$

lift-definition *uint64-of-int* :: $\langle \text{int} \Rightarrow \text{uint64} \rangle$
is *word-of-int* $\langle \text{proof} \rangle$

lift-definition *int-of-uint64* :: $\langle \text{uint64} \Rightarrow \text{int} \rangle$
is *uint* $\langle \text{proof} \rangle$

context
includes *integer.lifting*
begin

lift-definition *Uint64* :: $\langle \text{integer} \Rightarrow \text{uint64} \rangle$
is *word-of-int* $\langle \text{proof} \rangle$

lift-definition *integer-of-uint64* :: $\langle \text{uint64} \Rightarrow \text{integer} \rangle$
is *uint* $\langle \text{proof} \rangle$

end

global-interpretation *uint64*: *word-type-copy-more Abs-uint64 Rep-uint64 signed-drop-bit-uint64*
uint64-of-nat nat-of-uint64 uint64-of-int int-of-uint64 Uint64 integer-of-uint64
 $\langle \text{proof} \rangle$

instantiation *uint64* :: $\{ \text{size}, \text{msb}, \text{lsb}, \text{set-bit}, \text{bit-comprehension} \}$
begin

lift-definition *size-uint64* :: $\langle \text{uint64} \Rightarrow \text{nat} \rangle$ **is** *size* $\langle \text{proof} \rangle$

lift-definition *msb-uint64* :: $\langle \text{uint64} \Rightarrow \text{bool} \rangle$ **is** *msb* $\langle \text{proof} \rangle$

lift-definition *lsb-uint64* :: $\langle \text{uint64} \Rightarrow \text{bool} \rangle$ **is** *lsb* $\langle \text{proof} \rangle$

Workaround: avoid name space clash by spelling out *lift-definition* explicitly.

definition *set-bit-uint64* :: $\langle \text{uint64} \Rightarrow \text{nat} \Rightarrow \text{bool} \Rightarrow \text{uint64} \rangle$
where *set-bit-uint64-eq*: $\langle \text{set-bit-uint64 } a \ n \ b = (\text{if } b \ \text{then } \text{Bit-Operations.set-bit} \ \text{else } \text{unset-bit}) \ n \ a \rangle$

context
includes *lifting-syntax*
begin

lemma *set-bit-uint64-transfer* [*transfer-rule*]:
 $\langle (\text{cr-uint64} \ \text{====} \ (=) \ \text{====} \ (\leftarrow) \ \text{====} \ \text{cr-uint64}) \ \text{Generic-set-bit.set-bit} \ \text{Generic-set-bit.set-bit} \rangle$
 $\langle \text{proof} \rangle$

end

lift-definition *set-bits-uint64* :: $\langle (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{uint64} \rangle$ **is** *set-bits* $\langle \text{proof} \rangle$
lift-definition *set-bits-aux-uint64* :: $\langle (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow \text{uint64} \Rightarrow \text{uint64} \rangle$ **is**
set-bits-aux $\langle \text{proof} \rangle$

global-interpretation *uint64*: *word-type-copy-misc Abs-uint64 Rep-uint64 signed-drop-bit-uint64*
uint64-of-nat nat-of-uint64 uint64-of-int int-of-uint64 Uint64 integer-of-uint64 64
set-bits-aux-uint64
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

5.2 Code setup

For SML, we generate an implementation of unsigned 64-bit words using `IntInf.int`. If `LargeWord.wordSize > 63` of the Isabelle/ML runtime environment holds, then we assume that there is also a `Word64` structure available and accordingly replace the implementation for the target `Eval`.

code-printing code-module *Uint64* \rightarrow (SML) $\langle (* \text{ Test that words can handle numbers between 0 and 63 } *)$
val - = if 6 <= Word.wordSize then () else raise (Fail (wordSize less than 6));

```
structure Uint64 : sig
  eqtype uint64;
  val zero : uint64;
  val one : uint64;
  val fromInt : IntInf.int -> uint64;
  val toInt : uint64 -> IntInf.int;
  val toLarge : uint64 -> LargeWord.word;
  val fromLarge : LargeWord.word -> uint64
  val plus : uint64 -> uint64 -> uint64;
  val minus : uint64 -> uint64 -> uint64;
  val times : uint64 -> uint64 -> uint64;
  val divide : uint64 -> uint64 -> uint64;
  val modulus : uint64 -> uint64 -> uint64;
  val negate : uint64 -> uint64;
  val less-eq : uint64 -> uint64 -> bool;
  val less : uint64 -> uint64 -> bool;
  val notb : uint64 -> uint64;
  val andb : uint64 -> uint64 -> uint64;
  val orb : uint64 -> uint64 -> uint64;
  val xorb : uint64 -> uint64 -> uint64;
  val shifl : uint64 -> IntInf.int -> uint64;
  val shiftr : uint64 -> IntInf.int -> uint64;
  val shiftr-signed : uint64 -> IntInf.int -> uint64;
  val set-bit : uint64 -> IntInf.int -> bool -> uint64;
```

```

    val test-bit : uint64 -> IntInf.int -> bool;
end = struct

type uint64 = IntInf.int;

val mask = 0xFFFFFFFFFFFFFFFF : IntInf.int;

val zero = 0 : IntInf.int;

val one = 1 : IntInf.int;

fun fromInt x = IntInf.andb(x, mask);

fun toInt x = x

fun toLarge x = LargeWord.fromLargeInt (IntInf.toLarge x);

fun fromLarge x = IntInf.fromLarge (LargeWord.toLargeInt x);

fun plus x y = IntInf.andb(IntInf.+(x, y), mask);

fun minus x y = IntInf.andb(IntInf.-(x, y), mask);

fun negate x = IntInf.andb(IntInf.~(x), mask);

fun times x y = IntInf.andb(IntInf.*(x, y), mask);

fun divide x y = IntInf.div(x, y);

fun modulus x y = IntInf.mod(x, y);

fun less-eq x y = IntInf.<=(x, y);

fun less x y = IntInf.<(x, y);

fun notb x = IntInf.andb(IntInf.notb(x), mask);

fun orb x y = IntInf.orb(x, y);

fun andb x y = IntInf.andb(x, y);

fun xorb x y = IntInf.xorb(x, y);

val maxWord = IntInf.pow (2, Word.wordSize);

fun shiftl x n =
  if n < maxWord then IntInf.andb(IntInf.<<(x, Word.fromLargeInt (IntInf.toLarge
n)), mask)
  else 0;

```

```

fun shiftr x n =
  if n < maxWord then IntInf.~>> (x, Word.fromLargeInt (IntInf.toLarge n))
  else 0;

val msb-mask = 0x8000000000000000 : IntInf.int;

fun shiftr-signed x i =
  if IntInf.andb(x, msb-mask) = 0 then shiftr x i
  else if i >= 64 then 0xFFFFFFFFFFFFFFFF
  else let
    val x' = shiftr x i
    val m' = IntInf.andb(IntInf.<<(mask, Word.max(0w64 - Word.fromLargeInt
(IntInf.toLarge i), 0w0)), mask)
  in IntInf.orb(x', m') end;

fun test-bit x n =
  if n < maxWord then IntInf.andb (x, IntInf.<< (1, Word.fromLargeInt (IntInf.toLarge
n))) <> 0
  else false;

fun set-bit x n b =
  if n < 64 then
    if b then IntInf.orb (x, IntInf.<< (1, Word.fromLargeInt (IntInf.toLarge n)))
    else IntInf.andb (x, IntInf.notb (IntInf.<< (1, Word.fromLargeInt (IntInf.toLarge
n))))
  else x;

end
>
code-reserved SML UInt64

⟨ML⟩

code-printing code-module UInt64 → (Haskell)
⟨module UInt64 (Int64, Word64) where

  import Data.Int(Int64)
  import Data.Word(Word64)⟩
code-reserved Haskell UInt64

OCaml and Scala provide only signed 64bit numbers, so we use these and
implement sign-sensitive operations like comparisons manually.

code-printing code-module UInt64 → (OCaml)
⟨module UInt64 : sig
  val less : int64 → int64 → bool
  val less-eq : int64 → int64 → bool
  val set-bit : int64 → Z.t → bool → int64
  val shiftl : int64 → Z.t → int64

```

```

    val shiftr : int64 -> Z.t -> int64
    val shiftr-signed : int64 -> Z.t -> int64
    val test-bit : int64 -> Z.t -> bool
end = struct

(* negative numbers have their highest bit set,
   so they are greater than positive ones *)
let less x y =
  if Int64.compare x Int64.zero < 0 then
    Int64.compare y Int64.zero < 0 && Int64.compare x y < 0
  else Int64.compare y Int64.zero < 0 || Int64.compare x y < 0;;

let less-eq x y =
  if Int64.compare x Int64.zero < 0 then
    Int64.compare y Int64.zero < 0 && Int64.compare x y <= 0
  else Int64.compare y Int64.zero < 0 || Int64.compare x y <= 0;;

let set-bit x n b =
  let mask = Int64.shift-left Int64.one (Z.to-int n)
  in if b then Int64.logor x mask
     else Int64.logand x (Int64.lognot mask);;

let shiftl x n = Int64.shift-left x (Z.to-int n);;

let shiftr x n = Int64.shift-right-logical x (Z.to-int n);;

let shiftr-signed x n = Int64.shift-right x (Z.to-int n);;

let test-bit x n =
  Int64.compare
    (Int64.logand x (Int64.shift-left Int64.one (Z.to-int n)))
    Int64.zero
  <> 0;;

end;; (*struct Uint64*)
code-reserved OCaml Uint64

code-printing code-module Uint64 → (Scala)
⟨object Uint64 {

def less(x: Long, y: Long) : Boolean =
  x < 0 match {
    case true => y < 0 && x < y
    case false => y < 0 || x < y
  }

def less-eq(x: Long, y: Long) : Boolean =
  x < 0 match {
    case true => y < 0 && x <= y

```

```

    case false => y < 0 || x <= y
  }

def set-bit(x: Long, n: BigInt, b: Boolean) : Long =
  b match {
    case true => x | (1L << n.intValue)
    case false => x & (1L << n.intValue).unary~
  }

def shiftl(x: Long, n: BigInt) : Long = x << n.intValue

def shiftr(x: Long, n: BigInt) : Long = x >>> n.intValue

def shiftr-signed(x: Long, n: BigInt) : Long = x >> n.intValue

def test-bit(x: Long, n: BigInt) : Boolean =
  (x & (1L << n.intValue)) != 0

} /* object Uint64 */
code-reserved Scala Uint64

```

OCaml's conversion from `Big_int` to `int64` demands that the value fits into a signed 64-bit integer. The following justifies the implementation.

context

includes *bit-operations-syntax*

begin

definition *Uint64-signed* :: *integer* \Rightarrow *uint64*

where *Uint64-signed* *i* = (if *i* < $-(0x8000000000000000)$ \vee *i* \geq $0x8000000000000000$ then undefined *Uint64* *i* else *Uint64* *i*)

lemma *Uint64-code* [*code*]:

Uint64 *i* =
 (let *i'* = *i* AND $0xFFFFFFFFFFFFFFFF$
 in if bit *i'* 63 then *Uint64-signed* (*i'* - $0x10000000000000000$) else *Uint64-signed* *i'*)
including *undefined-transfer integer.lifting* <proof>

lemma *Uint64-signed-code* [*code*]:

Rep-uint64 (*Uint64-signed* *i*) =
 (if *i* < $-(0x8000000000000000)$ \vee *i* \geq $0x8000000000000000$ then *Rep-uint64*
 (undefined *Uint64* *i*) else *word-of-int* (*int-of-integer-symbolic* *i*)
 <proof>

end

Avoid *Abs-uint64* in generated code, use *Rep-uint64'* instead. The symbolic implementations for `code_simp` use *Rep-uint64*.

The new destructor *Rep-uint64'* is executable. As the simplifier is given the

[code abstract] equations literally, we cannot implement *Rep-uint64* directly, because that makes `code_simp` loop.

If code generation raises `Match`, some equation probably contains *Rep-uint64* ([code abstract] equations for *uint64* may use *Rep-uint64* because these instances will be folded away.)

To convert *64 word* values into *uint64*, use *Abs-uint64'*.

definition *Rep-uint64'* **where** [simp]: *Rep-uint64' = Rep-uint64*

lemma *Rep-uint64'-transfer* [transfer-rule]:
rel_fun cr_uint64 (=) (λx. x) Rep-uint64'
 ⟨proof⟩

lemma *Rep-uint64'-code* [code]: *Rep-uint64' x = (BITS n. bit x n)*
 ⟨proof⟩

lift-definition *Abs-uint64' :: 64 word ⇒ uint64* **is** *λx :: 64 word. x* ⟨proof⟩

lemma *Abs-uint64'-code* [code]:
Abs-uint64' x = Uint64 (integer-of-int (uint x))

including *integer.lifting* ⟨proof⟩

declare [[code drop: *term-of-class.term-of :: uint64 ⇒ -*]]

lemma *term-of-uint64-code* [code]:

defines *TR ≡ typerep.TypeRep* **and** *bit0 ≡ STR "Numeral-Type.bit0"*
shows
term-of-class.term-of x =
Code-Evaluation.App (Code-Evaluation.Const (STR "Uint64.uint64.Abs-uint64"))
(TR (STR "fun") [TR (STR "Word.word") [TR bit0 [TR bit0 [TR bit0 [TR bit0
[TR bit0 [TR bit0 [TR (STR "Numeral-Type.num1") []]]]]], TR (STR "Uint64.uint64")
[]])
(term-of-class.term-of (Rep-uint64' x))
 ⟨proof⟩

code-printing

type-constructor *uint64* \rightarrow
 (SML) *Uint64.uint64* **and**
 (Haskell) *Uint64.Word64* **and**
 (OCaml) *int64* **and**
 (Scala) *Long*
| **constant** *Uint64* \rightarrow
 (SML) *Uint64.fromInt* **and**
 (Haskell) (*Prelude.fromInteger - :: Uint64.Word64*) **and**
 (Haskell-Quickcheck) (*Prelude.fromInteger (Prelude.toInteger -) :: Uint64.Word64*)
and
 (Scala) *-.longValue*
| **constant** *Uint64-signed* \rightarrow
 (OCaml) *Z.to'-int64*

```

| constant 0 :: uint64 →
  (SML) Uint64.zero and
  (Haskell) (0 :: Uint64.Word64) and
  (OCaml) Int64.zero and
  (Scala) 0
| constant 1 :: uint64 →
  (SML) Uint64.one and
  (Haskell) (1 :: Uint64.Word64) and
  (OCaml) Int64.one and
  (Scala) 1
| constant plus :: uint64 ⇒ - →
  (SML) Uint64.plus and
  (Haskell) infixl 6 + and
  (OCaml) Int64.add and
  (Scala) infixl 7 +
| constant uminus :: uint64 ⇒ - →
  (SML) Uint64.negate and
  (Haskell) negate and
  (OCaml) Int64.neg and
  (Scala) !(-)
| constant minus :: uint64 ⇒ - →
  (SML) Uint64.minus and
  (Haskell) infixl 6 - and
  (OCaml) Int64.sub and
  (Scala) infixl 7 -
| constant times :: uint64 ⇒ - ⇒ - →
  (SML) Uint64.times and
  (Haskell) infixl 7 * and
  (OCaml) Int64.mul and
  (Scala) infixl 8 *
| constant HOL.equal :: uint64 ⇒ - ⇒ bool →
  (SML) !((- : Uint64.uint64) =) and
  (Haskell) infix 4 == and
  (OCaml) (Int64.compare - - = 0) and
  (Scala) infixl 5 ==
| class-instance uint64 :: equal →
  (Haskell) -
| constant less-eq :: uint64 ⇒ - ⇒ bool →
  (SML) Uint64.less'-eq and
  (Haskell) infix 4 <= and
  (OCaml) Uint64.less'-eq and
  (Scala) Uint64.less'-eq
| constant less :: uint64 ⇒ - ⇒ bool →
  (SML) Uint64.less and
  (Haskell) infix 4 < and
  (OCaml) Uint64.less and
  (Scala) Uint64.less
| constant Bit-Operations.not :: uint64 ⇒ - →
  (SML) Uint64.notb and

```

```

(Haskell) Data'-Bits.complement and
(OCaml) Int64.lognot and
(Scala) -.unary'~
| constant Bit-Operations.and :: uint64 ⇒ - →
(SML) Uint64.andb and
(Haskell) infixl 7 Data-Bits.&. and
(OCaml) Int64.logand and
(Scala) infixl 3 &
| constant Bit-Operations.or :: uint64 ⇒ - →
(SML) Uint64.orb and
(Haskell) infixl 5 Data-Bits.|. and
(OCaml) Int64.logor and
(Scala) infixl 1 |
| constant Bit-Operations.xor :: uint64 ⇒ - →
(SML) Uint64.xorb and
(Haskell) Data'-Bits.xor and
(OCaml) Int64.logxor and
(Scala) infixl 2 ^

definition uint64-divmod :: uint64 ⇒ uint64 ⇒ uint64 × uint64 where
  uint64-divmod x y =
    (if y = 0 then (undefined ((div) :: uint64 ⇒ -) x (0 :: uint64), undefined ((mod)
:: uint64 ⇒ -) x (0 :: uint64))
    else (x div y, x mod y))

definition uint64-div :: uint64 ⇒ uint64 ⇒ uint64
where uint64-div x y = fst (uint64-divmod x y)

definition uint64-mod :: uint64 ⇒ uint64 ⇒ uint64
where uint64-mod x y = snd (uint64-divmod x y)

lemma div-uint64-code [code]: x div y = (if y = 0 then 0 else uint64-div x y)
including undefined-transfer ⟨proof⟩

lemma mod-uint64-code [code]: x mod y = (if y = 0 then x else uint64-mod x y)
including undefined-transfer ⟨proof⟩

definition uint64-sdiv :: uint64 ⇒ uint64 ⇒ uint64
where [code del]:
  uint64-sdiv x y =
    (if y = 0 then undefined ((div) :: uint64 ⇒ -) x (0 :: uint64)
    else Abs-uint64 (Rep-uint64 x sdiv Rep-uint64 y))

definition div0-uint64 :: uint64 ⇒ uint64
where [code del]: div0-uint64 x = undefined ((div) :: uint64 ⇒ -) x (0 :: uint64)
declare [[code abort: div0-uint64]]

definition mod0-uint64 :: uint64 ⇒ uint64
where [code del]: mod0-uint64 x = undefined ((mod) :: uint64 ⇒ -) x (0 :: uint64)

```

declare `[[code abort: mod0-uint64]]`

lemma `uint64-divmod-code [code]:`

```
uint64-divmod x y =
  (if 0x8000000000000000 ≤ y then if x < y then (0, x) else (1, x - y)
   else if y = 0 then (div0-uint64 x, mod0-uint64 x)
   else let q = push-bit 1 (uint64-sdiv (drop-bit 1 x) y);
        r = x - q * y
        in if r ≥ y then (q + 1, r - y) else (q, r))
```

including `undefined-transfer <proof>`

lemma `uint64-sdiv-code [code]:`

```
Rep-uint64 (uint64-sdiv x y) =
  (if y = 0 then Rep-uint64 (undefined ((div) :: uint64 ⇒ -) x (0 :: uint64))
   else Rep-uint64 x sdiv Rep-uint64 y)
```

`<proof>`

Note that we only need a translation for signed division, but not for the remainder because `uint64-divmod ?x ?y = (if 9223372036854775808 ≤ ?y then if ?x < ?y then (0, ?x) else (1, ?x - ?y) else if ?y = 0 then (div0-uint64 ?x, mod0-uint64 ?x) else let q = push-bit 1 (uint64-sdiv (drop-bit 1 ?x) ?y); r = ?x - q * ?y in if ?y ≤ r then (q + 1, r - ?y) else (q, r))` computes both with division only.

code-printing

```
constant uint64-div →
  (SML) Uint64.divide and
  (Haskell) Prelude.div
| constant uint64-mod →
  (SML) Uint64.modulus and
  (Haskell) Prelude.mod
| constant uint64-divmod →
  (Haskell) divmod
| constant uint64-sdiv →
  (OCaml) Int64.div and
  (Scala) - '/' -
```

definition `uint64-test-bit :: uint64 ⇒ integer ⇒ bool`

where `[code del]:`

```
uint64-test-bit x n =
  (if n < 0 ∨ 63 < n then undefined (bit :: uint64 ⇒ -) x n
   else bit x (nat-of-integer n))
```

lemma `bit-uint64-code [code]:`

```
bit x n ↔ n < 64 ∧ uint64-test-bit x (integer-of-nat n)
including undefined-transfer integer.lifting <proof>
```

lemma `uint64-test-bit-code [code]:`

```
uint64-test-bit w n =
  (if n < 0 ∨ 63 < n then undefined (bit :: uint64 ⇒ -) w n else bit (Rep-uint64
```

w) (*nat-of-integer* n))
 ⟨*proof*⟩

code-printing constant *uint64-test-bit* \rightarrow
 (SML) *Uint64.test'-bit* **and**
 (Haskell) *Data'-Bits.testBitBounded* **and**
 (OCaml) *Uint64.test'-bit* **and**
 (Scala) *Uint64.test'-bit* **and**
 (Eval) (*fn* x => *fn* i => if i < 0 orelse i >= 64 then raise (Fail argument to *uint64'-test'-bit* out of bounds) else *Uint64.test'-bit* x i)

definition *uint64-set-bit* :: *uint64* \Rightarrow *integer* \Rightarrow *bool* \Rightarrow *uint64*

where [*code del*]:

uint64-set-bit x n b =
 (if n < 0 \vee 63 < n then undefined (*set-bit* :: *uint64* \Rightarrow -) x n b
 else *set-bit* x (*nat-of-integer* n) b)

lemma *set-bit-uint64-code* [*code*]:

set-bit x n b = (if n < 64 then *uint64-set-bit* x (*integer-of-nat* n) b else x)

including *undefined-transfer integer.lifting* ⟨*proof*⟩

lemma *uint64-set-bit-code* [*code*]:

Rep-uint64 (*uint64-set-bit* w n b) =
 (if n < 0 \vee 63 < n then *Rep-uint64* (undefined (*set-bit* :: *uint64* \Rightarrow -) w n b)
 else *set-bit* (*Rep-uint64* w) (*nat-of-integer* n) b)

including *undefined-transfer* ⟨*proof*⟩

code-printing constant *uint64-set-bit* \rightarrow

(SML) *Uint64.set'-bit* **and**
 (Haskell) *Data'-Bits.setBitBounded* **and**
 (OCaml) *Uint64.set'-bit* **and**
 (Scala) *Uint64.set'-bit* **and**
 (Eval) (*fn* x => *fn* i => *fn* b => if i < 0 orelse i >= 64 then raise (Fail argument to *uint64'-set'-bit* out of bounds) else *Uint64.set'-bit* x i b)

definition *uint64-shiffl* :: *uint64* \Rightarrow *integer* \Rightarrow *uint64*

where [*code del*]:

uint64-shiffl x n = (if n < 0 \vee 64 \leq n then undefined (*push-bit* :: *nat* \Rightarrow *uint64* \Rightarrow -) x n else *push-bit* (*nat-of-integer* n) x)

lemma *shiffl-uint64-code* [*code*]: *push-bit* n x = (if n < 64 then *uint64-shiffl* x (*integer-of-nat* n) else 0)

including *undefined-transfer integer.lifting* ⟨*proof*⟩

lemma *uint64-shiffl-code* [*code*]:

Rep-uint64 (*uint64-shiffl* w n) =
 (if n < 0 \vee 64 \leq n then *Rep-uint64* (undefined (*push-bit* :: *nat* \Rightarrow *uint64* \Rightarrow -) w n) else *push-bit* (*nat-of-integer* n) (*Rep-uint64* w))

including *undefined-transfer* ⟨*proof*⟩

code-printing constant *uint64-shiffl* \rightarrow
 (SML) *Uint64.shiffl* **and**
 (Haskell) *Data'-Bits.shifflBounded* **and**
 (OCaml) *Uint64.shiffl* **and**
 (Scala) *Uint64.shiffl* **and**
 (Eval) (fn x => fn i => if i < 0 orelse i >= 64 then raise (Fail argument to uint64'-shiffl out of bounds) else Uint64.shiffl x i)

definition *uint64-shiftr* :: *uint64* \Rightarrow *integer* \Rightarrow *uint64*
where [code del]:
uint64-shiftr x n = (if n < 0 \vee 64 \leq n then undefined (drop-bit :: nat \Rightarrow uint64 \Rightarrow -) x n else drop-bit (nat-of-integer n) x)

lemma *shiftr-uint64-code* [code]: drop-bit n x = (if n < 64 then *uint64-shiftr* x (integer-of-nat n) else 0)
including *undefined-transfer integer.lifting* <proof>

lemma *uint64-shiftr-code* [code]:
Rep-uint64 (*uint64-shiftr* w n) =
 (if n < 0 \vee 64 \leq n then *Rep-uint64* (undefined (drop-bit :: nat \Rightarrow uint64 \Rightarrow -) w n) else drop-bit (nat-of-integer n) (*Rep-uint64* w))
including *undefined-transfer* <proof>

code-printing constant *uint64-shiftr* \rightarrow
 (SML) *Uint64.shiftr* **and**
 (Haskell) *Data'-Bits.shiftrBounded* **and**
 (OCaml) *Uint64.shiftr* **and**
 (Scala) *Uint64.shiftr* **and**
 (Eval) (fn x => fn i => if i < 0 orelse i >= 64 then raise (Fail argument to uint64'-shiftr out of bounds) else Uint64.shiftr x i)

definition *uint64-sshiftr* :: *uint64* \Rightarrow *integer* \Rightarrow *uint64*
where [code del]:
uint64-sshiftr x n =
 (if n < 0 \vee 64 \leq n then undefined *signed-drop-bit-uint64* n x else *signed-drop-bit-uint64* (nat-of-integer n) x)

lemma *sshiftr-uint64-code* [code]:
signed-drop-bit-uint64 n x =
 (if n < 64 then *uint64-sshiftr* x (integer-of-nat n) else if bit x 63 then - 1 else 0)
including *undefined-transfer integer.lifting* <proof>

lemma *uint64-sshiftr-code* [code]:
Rep-uint64 (*uint64-sshiftr* w n) =
 (if n < 0 \vee 64 \leq n then *Rep-uint64* (undefined *signed-drop-bit-uint64* n w) else *signed-drop-bit* (nat-of-integer n) (*Rep-uint64* w))
including *undefined-transfer* <proof>

code-printing constant *uint64-sshiftr* \rightarrow
 (SML) *Uint64.shiftr'-signed* **and**
 (Haskell)
 (*Prelude.fromInteger (Prelude.toInteger (Data'-Bits.shiftrBounded (Prelude.fromInteger (Prelude.toInteger -) :: Uint64.Int64) -)) :: Uint64.Word64*) **and**
 (OCaml) *Uint64.shiftr'-signed* **and**
 (Scala) *Uint64.shiftr'-signed* **and**
 (Eval) (*fn x => fn i => if i < 0 orelse i >= 64 then raise (Fail argument to uint64'-shiftr'-signed out of bounds) else Uint64.shiftr'-signed x i*)

context

includes *bit-operations-syntax*
begin

lemma *uint64-msb-test-bit*: $msb\ x \longleftrightarrow bit\ (x :: uint64)\ 63$
<proof>

lemma *msb-uint64-code* [code]: $msb\ x \longleftrightarrow uint64-test-bit\ x\ 63$
<proof>

lemma *uint64-of-int-code* [code]:
uint64-of-int i = Uint64 (integer-of-int i)
including *integer.lifting* *<proof>*

lemma *int-of-uint64-code* [code]:
int-of-uint64 x = int-of-integer (integer-of-uint64 x)
including *integer.lifting* *<proof>*

lemma *uint64-of-nat-code* [code]:
uint64-of-nat = uint64-of-int \circ int
<proof>

lemma *nat-of-uint64-code* [code]:
nat-of-uint64 x = nat-of-integer (integer-of-uint64 x)
<proof> **including** *integer.lifting* *<proof>*

definition *integer-of-uint64-signed* :: *uint64* \Rightarrow *integer*

where

integer-of-uint64-signed n = (if bit n 63 then undefined integer-of-uint64 n else integer-of-uint64 n)

lemma *integer-of-uint64-signed-code* [code]:
integer-of-uint64-signed n =
(if bit n 63 then undefined integer-of-uint64 n else integer-of-int (uint (Rep-uint64' n)))
<proof>

lemma *integer-of-uint64-code* [code]:
integer-of-uint64 n =

```

    (if bit n 63 then integer-of-wint64-signed (n AND 0x7FFFFFFFFFFFFFFF) OR
     0x8000000000000000 else integer-of-wint64-signed n)
  <proof>
    including integer.lifting <proof>

end

```

code-printing

```

  constant integer-of-uint64  $\rightarrow$ 
    (SML) Uint64.toInt and
    (Haskell) Prelude.toInteger
| constant integer-of-wint64-signed  $\rightarrow$ 
    (OCaml) Z.of'-int64 and
    (Scala) BigInt

```

5.3 Quickcheck setup

```

definition wint64-of-natural :: natural  $\Rightarrow$  uint64
where wint64-of-natural x  $\equiv$  Uint64 (integer-of-natural x)

```

```

instantiation uint64 :: {random, exhaustive, full-exhaustive} begin

```

```

definition random-uint64  $\equiv$  qc-random-cnv uint64-of-natural

```

```

definition exhaustive-uint64  $\equiv$  qc-exhaustive-cnv uint64-of-natural

```

```

definition full-exhaustive-uint64  $\equiv$  qc-full-exhaustive-cnv uint64-of-natural

```

```

instance <proof>

```

```

end

```

```

instantiation uint64 :: narrowing begin

```

```

interpretation quickcheck-narrowing-samples

```

```

   $\lambda i$ . let x = Uint64 i in (x, 0xFFFFFFFFFFFFFFFF - x) 0

```

```

  Typerep.Typerep (STR "Uint64.uint64") [] <proof>

```

```

definition narrowing-uint64 d = qc-narrowing-drawn-from (narrowing-samples d)
d

```

```

declare [[code drop: partial-term-of :: uint64 itself  $\Rightarrow$  -]]

```

```

lemmas partial-term-of-uint64 [code] = partial-term-of-code

```

```

instance <proof>

```

```

end

```

```

end

```


Chapter 6

Unsigned words of 32 bits

```
theory Uint32 imports
  Word-Type-Copies
  Code-Target-Integer-Bit
begin
```

6.1 Type definition and primitive operations

```
typedef uint32 = ⟨UNIV :: 32 word set⟩ ⟨proof⟩
```

```
global-interpretation uint32: word-type-copy Abs-uint32 Rep-uint32
  ⟨proof⟩
```

```
setup-lifting type-definition-uint32
```

```
declare uint32.of-word-of [code abstype]
```

```
declare Quotient-uint32 [transfer-rule]
```

```
instantiation uint32 :: ⟨{comm-ring-1, semiring-modulo, equal, linorder}⟩
begin
```

```
lift-definition zero-uint32 :: uint32 is 0 ⟨proof⟩
```

```
lift-definition one-uint32 :: uint32 is 1 ⟨proof⟩
```

```
lift-definition plus-uint32 :: ⟨uint32 ⇒ uint32 ⇒ uint32⟩ is ⟨(+)⟩ ⟨proof⟩
```

```
lift-definition uminus-uint32 :: ⟨uint32 ⇒ uint32⟩ is uminus ⟨proof⟩
```

```
lift-definition minus-uint32 :: ⟨uint32 ⇒ uint32 ⇒ uint32⟩ is ⟨(-)⟩ ⟨proof⟩
```

```
lift-definition times-uint32 :: ⟨uint32 ⇒ uint32 ⇒ uint32⟩ is ⟨(*)⟩ ⟨proof⟩
```

```
lift-definition divide-uint32 :: ⟨uint32 ⇒ uint32 ⇒ uint32⟩ is ⟨(div)⟩ ⟨proof⟩
```

```
lift-definition modulo-uint32 :: ⟨uint32 ⇒ uint32 ⇒ uint32⟩ is ⟨(mod)⟩ ⟨proof⟩
```

```
lift-definition equal-uint32 :: ⟨uint32 ⇒ uint32 ⇒ bool⟩ is ⟨HOL.equal⟩ ⟨proof⟩
```

```
lift-definition less-eq-uint32 :: ⟨uint32 ⇒ uint32 ⇒ bool⟩ is ⟨(≤)⟩ ⟨proof⟩
```

```
lift-definition less-uint32 :: ⟨uint32 ⇒ uint32 ⇒ bool⟩ is ⟨(<)⟩ ⟨proof⟩
```

```
global-interpretation uint32: word-type-copy-ring Abs-uint32 Rep-uint32
```

```

  <proof>

instance <proof>

end

instantiation uint32 :: ring-bit-operations
begin

lift-definition bit-uint32 :: <uint32 ⇒ nat ⇒ bool> is bit <proof>
lift-definition not-uint32 :: <uint32 ⇒ uint32> is <Bit-Operations.not> <proof>
lift-definition and-uint32 :: <uint32 ⇒ uint32 ⇒ uint32> is <Bit-Operations.and>
<proof>
lift-definition or-uint32 :: <uint32 ⇒ uint32 ⇒ uint32> is <Bit-Operations.or>
<proof>
lift-definition xor-uint32 :: <uint32 ⇒ uint32 ⇒ uint32> is <Bit-Operations.xor>
<proof>
lift-definition mask-uint32 :: <nat ⇒ uint32> is mask <proof>
lift-definition push-bit-uint32 :: <nat ⇒ uint32 ⇒ uint32> is push-bit <proof>
lift-definition drop-bit-uint32 :: <nat ⇒ uint32 ⇒ uint32> is drop-bit <proof>
lift-definition signed-drop-bit-uint32 :: <nat ⇒ uint32 ⇒ uint32> is signed-drop-bit
<proof>
lift-definition take-bit-uint32 :: <nat ⇒ uint32 ⇒ uint32> is take-bit <proof>
lift-definition set-bit-uint32 :: <nat ⇒ uint32 ⇒ uint32> is Bit-Operations.set-bit
<proof>
lift-definition unset-bit-uint32 :: <nat ⇒ uint32 ⇒ uint32> is unset-bit <proof>
lift-definition flip-bit-uint32 :: <nat ⇒ uint32 ⇒ uint32> is flip-bit <proof>

global-interpretation uint32: word-type-copy-bits Abs-uint32 Rep-uint32 signed-drop-bit-uint32
<proof>

instance
  <proof>

end

lift-definition uint32-of-nat :: <nat ⇒ uint32>
  is word-of-nat <proof>

lift-definition nat-of-uint32 :: <uint32 ⇒ nat>
  is unat <proof>

lift-definition uint32-of-int :: <int ⇒ uint32>
  is word-of-int <proof>

lift-definition int-of-uint32 :: <uint32 ⇒ int>
  is uint <proof>

context

```

```

includes integer.lifting
begin

```

```

lift-definition Uint32 ::  $\langle \text{integer} \Rightarrow \text{uint32} \rangle$ 
  is word-of-int  $\langle \text{proof} \rangle$ 

```

```

lift-definition integer-of-uint32 ::  $\langle \text{uint32} \Rightarrow \text{integer} \rangle$ 
  is uint  $\langle \text{proof} \rangle$ 

```

```

end

```

```

global-interpretation uint32: word-type-copy-more Abs-uint32 Rep-uint32 signed-drop-bit-uint32
  uint32-of-nat nat-of-uint32 uint32-of-int int-of-uint32 Uint32 integer-of-uint32
   $\langle \text{proof} \rangle$ 

```

```

instantiation uint32 :: {size, msb, lsb, set-bit, bit-comprehension}
begin

```

```

lift-definition size-uint32 ::  $\langle \text{uint32} \Rightarrow \text{nat} \rangle$  is size  $\langle \text{proof} \rangle$ 

```

```

lift-definition msb-uint32 ::  $\langle \text{uint32} \Rightarrow \text{bool} \rangle$  is msb  $\langle \text{proof} \rangle$ 

```

```

lift-definition lsb-uint32 ::  $\langle \text{uint32} \Rightarrow \text{bool} \rangle$  is lsb  $\langle \text{proof} \rangle$ 

```

Workaround: avoid name space clash by spelling out *lift-definition* explicitly.

```

definition set-bit-uint32 ::  $\langle \text{uint32} \Rightarrow \text{nat} \Rightarrow \text{bool} \Rightarrow \text{uint32} \rangle$ 
  where set-bit-uint32-eq:  $\langle \text{set-bit-uint32 } a \ n \ b = (\text{if } b \ \text{then } \text{Bit-Operations.set-bit} \ \text{else } \text{unset-bit}) \ n \ a \rangle$ 

```

```

context
  includes lifting-syntax
begin

```

```

lemma set-bit-uint32-transfer [transfer-rule]:
   $\langle (\text{cr-uint32} \ \text{====} \ (=) \ \text{====} \ (\longleftrightarrow) \ \text{====} \ \text{cr-uint32}) \ \text{Generic-set-bit.set-bit} \ \text{Generic-set-bit.set-bit} \rangle$ 
   $\langle \text{proof} \rangle$ 

```

```

end

```

```

lift-definition set-bits-uint32 ::  $\langle (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{uint32} \rangle$  is set-bits  $\langle \text{proof} \rangle$ 

```

```

lift-definition set-bits-aux-uint32 ::  $\langle (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow \text{uint32} \Rightarrow \text{uint32} \rangle$  is
  set-bits-aux  $\langle \text{proof} \rangle$ 

```

```

global-interpretation uint32: word-type-copy-misc Abs-uint32 Rep-uint32 signed-drop-bit-uint32
  uint32-of-nat nat-of-uint32 uint32-of-int int-of-uint32 Uint32 integer-of-uint32 32
  set-bits-aux-uint32
   $\langle \text{proof} \rangle$ 

```

```

instance  $\langle \text{proof} \rangle$ 

```

end

6.2 Code setup

```

code-printing code-module Uint32  $\rightarrow$  (SML)
⟨(* Test that words can handle numbers between 0 and 31 *)
val - = if 5 <= Word.wordSize then () else raise (Fail (wordSize less than 5));

structure Uint32 : sig
  val set-bit : Word32.word  $\rightarrow$  IntInf.int  $\rightarrow$  bool  $\rightarrow$  Word32.word
  val shiftl : Word32.word  $\rightarrow$  IntInf.int  $\rightarrow$  Word32.word
  val shiftr : Word32.word  $\rightarrow$  IntInf.int  $\rightarrow$  Word32.word
  val shiftr-signed : Word32.word  $\rightarrow$  IntInf.int  $\rightarrow$  Word32.word
  val test-bit : Word32.word  $\rightarrow$  IntInf.int  $\rightarrow$  bool
end = struct

fun set-bit x n b =
  let val mask = Word32.<<< (0wx1, Word.fromLargeInt (IntInf.toLarge n))
  in if b then Word32.orb (x, mask)
    else Word32.andb (x, Word32.notb mask)
  end

fun shiftl x n =
  Word32.<<< (x, Word.fromLargeInt (IntInf.toLarge n))

fun shiftr x n =
  Word32.>>> (x, Word.fromLargeInt (IntInf.toLarge n))

fun shiftr-signed x n =
  Word32.~>>> (x, Word.fromLargeInt (IntInf.toLarge n))

fun test-bit x n =
  Word32.andb (x, Word32.<<< (0wx1, Word.fromLargeInt (IntInf.toLarge n)))
  <> Word32.fromInt 0

end; (* struct Uint32 *)
code-reserved SML Uint32

code-printing code-module Uint32  $\rightarrow$  (Haskell)
⟨module Uint32(Int32, Word32) where

  import Data.Int(Int32)
  import Data.Word(Word32)
code-reserved Haskell Uint32

OCaml and Scala provide only signed 32bit numbers, so we use these and
implement sign-sensitive operations like comparisons manually.

code-printing code-module Uint32  $\rightarrow$  (OCaml)

```

```

⟨module Uint32 : sig
  val less : int32 -> int32 -> bool
  val less-eq : int32 -> int32 -> bool
  val set-bit : int32 -> Z.t -> bool -> int32
  val shiftl : int32 -> Z.t -> int32
  val shiftr : int32 -> Z.t -> int32
  val shiftr-signed : int32 -> Z.t -> int32
  val test-bit : int32 -> Z.t -> bool
end = struct

(* negative numbers have their highest bit set,
   so they are greater than positive ones *)
let less x y =
  if Int32.compare x Int32.zero < 0 then
    Int32.compare y Int32.zero < 0 && Int32.compare x y < 0
  else Int32.compare y Int32.zero < 0 || Int32.compare x y < 0;;

let less-eq x y =
  if Int32.compare x Int32.zero < 0 then
    Int32.compare y Int32.zero < 0 && Int32.compare x y <= 0
  else Int32.compare y Int32.zero < 0 || Int32.compare x y <= 0;;

let set-bit x n b =
  let mask = Int32.shift-left Int32.one (Z.to-int n)
  in if b then Int32.logor x mask
     else Int32.logand x (Int32.lognot mask);;

let shiftl x n = Int32.shift-left x (Z.to-int n);;

let shiftr x n = Int32.shift-right-logical x (Z.to-int n);;

let shiftr-signed x n = Int32.shift-right x (Z.to-int n);;

let test-bit x n =
  Int32.compare
    (Int32.logand x (Int32.shift-left Int32.one (Z.to-int n)))
    Int32.zero
  <> 0;;

end;; (*struct Uint32*)
code-reserved OCaml Uint32

code-printing code-module Uint32 → (Scala)
⟨object Uint32 {

def less(x: Int, y: Int) : Boolean =
  x < 0 match {
    case true => y < 0 && x < y
    case false => y < 0 || x < y
  }
}

```

```

}

def less-eq(x: Int, y: Int) : Boolean =
  x < 0 match {
    case true => y < 0 && x <= y
    case false => y < 0 || x <= y
  }

def set-bit(x: Int, n: BigInt, b: Boolean) : Int =
  b match {
    case true => x | (1 << n.toIntValue)
    case false => x & (1 << n.toIntValue).unary~
  }

def shiftl(x: Int, n: BigInt) : Int = x << n.toIntValue

def shiftr(x: Int, n: BigInt) : Int = x >>> n.toIntValue

def shiftr-signed(x: Int, n: BigInt) : Int = x >> n.toIntValue

def test-bit(x: Int, n: BigInt) : Boolean =
  (x & (1 << n.toIntValue)) != 0

} /* object Uint32 */
code-reserved Scala Uint32

```

OCaml's conversion from `Big_int` to `int32` demands that the value fits into a signed 32-bit integer. The following justifies the implementation.

context

includes `bit-operations-syntax`

begin

definition `Uint32-signed` :: `integer` \Rightarrow `wint32`

where `Uint32-signed` i = (if $i < -(0x80000000) \vee i \geq 0x80000000$ then `undefined` `Uint32` i else `Uint32` i)

lemma `Uint32-code` [`code`]:

`Uint32` i =
 (let $i' = i$ AND `0xFFFFFFFF`
 in if bit i' 31 then `Uint32-signed` ($i' - 0x100000000$) else `Uint32-signed` i')
including `undefined-transfer` `integer.lifting` \langle proof \rangle

lemma `Uint32-signed-code` [`code`]:

`Rep-wint32` (`Uint32-signed` i) =
 (if $i < -(0x80000000) \vee i \geq 0x80000000$ then `Rep-wint32` (`undefined` `Uint32` i)
 else `word-of-int` (`int-of-integer-symbolic` i)
 \langle proof \rangle

end

Avoid *Abs-uint32* in generated code, use *Rep-uint32'* instead. The symbolic implementations for `code_simp` use *Rep-uint32*.

The new destructor *Rep-uint32'* is executable. As the simplifier is given the `[code abstract]` equations literally, we cannot implement *Rep-uint32* directly, because that makes `code_simp` loop.

If code generation raises `Match`, some equation probably contains *Rep-uint32* (`[code abstract]` equations for *uint32* may use *Rep-uint32* because these instances will be folded away.)

To convert *32 word* values into *uint32*, use *Abs-uint32'*.

definition *Rep-uint32'* **where** `[simp]: Rep-uint32' = Rep-uint32`

lemma *Rep-uint32'-transfer* `[transfer-rule]:`
`rel_fun cr_uint32 (=) (λx. x) Rep-uint32'`
`<proof>`

lemma *Rep-uint32'-code* `[code]: Rep-uint32' x = (BITS n. bit x n)`
`<proof>`

lift-definition *Abs-uint32'* `:: 32 word ⇒ uint32` **is** `λx :: 32 word. x` `<proof>`

lemma *Abs-uint32'-code* `[code]:`
`Abs-uint32' x = Uint32 (integer-of-int (uint x))`

including *integer.lifting* `<proof>`

declare `[[code drop: term-of-class.term-of :: uint32 ⇒ -]]`

lemma *term-of-uint32-code* `[code]:`

defines `TR ≡ typerep.TypeRep` **and** `bit0 ≡ STR "Numeral-Type.bit0"`
shows
`term-of-class.term-of x =`
`Code-Evaluation.App (Code-Evaluation.Const (STR "Uint32.uint32.Abs-uint32"))`
`(TR (STR "fun") [TR (STR "Word.word") [TR bit0 [TR bit0 [TR bit0 [TR bit0`
`[TR bit0 [TR (STR "Numeral-Type.num1") []]]]]], TR (STR "Uint32.uint32")`
`[]))`
`(term-of-class.term-of (Rep-uint32' x))`
`<proof>`

code-printing

type-constructor *uint32* `↔`
`(SML) Word32.word` **and**
`(Haskell) Uint32.Word32` **and**
`(OCaml) int32` **and**
`(Scala) Int` **and**
`(Eval) Word32.word`
| **constant** *Uint32* `↔`
`(SML) Word32.fromLargeInt (IntInf.toLarge -)` **and**
`(Haskell) (Prelude.fromInteger - :: Uint32.Word32)` **and**

```

(Haskell-Quickcheck) (Prelude.fromInteger (Prelude.toInteger -) :: Uint32.Word32)
and
  (Scala) -.intValue
| constant Uint32-signed →
  (OCaml) Z.to'-int32
| constant 0 :: uint32 →
  (SML) (Word32.fromInt 0) and
  (Haskell) (0 :: Uint32.Word32) and
  (OCaml) Int32.zero and
  (Scala) 0
| constant 1 :: uint32 →
  (SML) (Word32.fromInt 1) and
  (Haskell) (1 :: Uint32.Word32) and
  (OCaml) Int32.one and
  (Scala) 1
| constant plus :: uint32 ⇒ - →
  (SML) Word32.+ ((-), (-)) and
  (Haskell) infixl 6 + and
  (OCaml) Int32.add and
  (Scala) infixl 7 +
| constant uminus :: uint32 ⇒ - →
  (SML) Word32.~ and
  (Haskell) negate and
  (OCaml) Int32.neg and
  (Scala) !(-)
| constant minus :: uint32 ⇒ - →
  (SML) Word32.- ((-), (-)) and
  (Haskell) infixl 6 - and
  (OCaml) Int32.sub and
  (Scala) infixl 7 -
| constant times :: uint32 ⇒ - ⇒ - →
  (SML) Word32.* ((-), (-)) and
  (Haskell) infixl 7 * and
  (OCaml) Int32.mul and
  (Scala) infixl 8 *
| constant HOL.equal :: uint32 ⇒ - ⇒ bool →
  (SML) !((- : Word32.word) = -) and
  (Haskell) infix 4 == and
  (OCaml) (Int32.compare - - = 0) and
  (Scala) infixl 5 ==
| class-instance uint32 :: equal →
  (Haskell) -
| constant less-eq :: uint32 ⇒ - ⇒ bool →
  (SML) Word32.<= ((-), (-)) and
  (Haskell) infix 4 <= and
  (OCaml) Uint32.less'-eq and
  (Scala) Uint32.less'-eq
| constant less :: uint32 ⇒ - ⇒ bool →
  (SML) Word32.< ((-), (-)) and

```



```

(Haskell) infix 4 < and
(OCaml) Uint32.less and
(Scala) Uint32.less
| constant Bit-Operations.not :: uint32 ⇒ - →
(SML) Word32.notb and
(Haskell) Data'-Bits.complement and
(OCaml) Int32.lognot and
(Scala) -.unary'~
| constant Bit-Operations.and :: uint32 ⇒ - →
(SML) Word32.andb ((-),/ (-)) and
(Haskell) infixl 7 Data'-Bits.& and
(OCaml) Int32.logand and
(Scala) infixl 3 &
| constant Bit-Operations.or :: uint32 ⇒ - →
(SML) Word32.orb ((-),/ (-)) and
(Haskell) infixl 5 Data'-Bits.|. and
(OCaml) Int32.logor and
(Scala) infixl 1 |
| constant Bit-Operations.xor :: uint32 ⇒ - →
(SML) Word32.xorb ((-),/ (-)) and
(Haskell) Data'-Bits.xor and
(OCaml) Int32.logxor and
(Scala) infixl 2 ^

definition uint32-divmod :: uint32 ⇒ uint32 ⇒ uint32 × uint32 where
  uint32-divmod x y =
    (if y = 0 then (undefined ((div) :: uint32 ⇒ -) x (0 :: uint32), undefined ((mod)
:: uint32 ⇒ -) x (0 :: uint32))
    else (x div y, x mod y))

definition uint32-div :: uint32 ⇒ uint32 ⇒ uint32
where uint32-div x y = fst (uint32-divmod x y)

definition uint32-mod :: uint32 ⇒ uint32 ⇒ uint32
where uint32-mod x y = snd (uint32-divmod x y)

lemma div-uint32-code [code]: x div y = (if y = 0 then 0 else uint32-div x y)
including undefined-transfer ⟨proof⟩

lemma mod-uint32-code [code]: x mod y = (if y = 0 then x else uint32-mod x y)
including undefined-transfer ⟨proof⟩

definition uint32-sdiv :: uint32 ⇒ uint32 ⇒ uint32
where [code del]:
  uint32-sdiv x y =
    (if y = 0 then undefined ((div) :: uint32 ⇒ -) x (0 :: uint32)
    else Abs-uint32 (Rep-uint32 x sdiv Rep-uint32 y))

definition div0-uint32 :: uint32 ⇒ uint32

```

where `[code del]: div0-uint32 x = undefined ((div) :: uint32 ⇒ -) x (0 :: uint32)`
declare `[[code abort: div0-uint32]]`

definition `mod0-uint32 :: uint32 ⇒ uint32`
where `[code del]: mod0-uint32 x = undefined ((mod) :: uint32 ⇒ -) x (0 :: uint32)`
declare `[[code abort: mod0-uint32]]`

lemma `uint32-divmod-code [code]:`
`uint32-divmod x y =`
`(if 0x80000000 ≤ y then if x < y then (0, x) else (1, x - y)`
`else if y = 0 then (div0-uint32 x, mod0-uint32 x)`
`else let q = push-bit 1 (uint32-sdiv (drop-bit 1 x) y);`
`r = x - q * y`
`in if r ≥ y then (q + 1, r - y) else (q, r))`
including `undefined-transfer <proof>`

lemma `uint32-sdiv-code [code]:`
`Rep-uint32 (uint32-sdiv x y) =`
`(if y = 0 then Rep-uint32 (undefined ((div) :: uint32 ⇒ -) x (0 :: uint32))`
`else Rep-uint32 x sdiv Rep-uint32 y)`
`<proof>`

Note that we only need a translation for signed division, but not for the remainder because `uint32-divmod ?x ?y = (if 2147483648 ≤ ?y then if ?x < ?y then (0, ?x) else (1, ?x - ?y) else if ?y = 0 then (div0-uint32 ?x, mod0-uint32 ?x) else let q = push-bit 1 (uint32-sdiv (drop-bit 1 ?x) ?y); r = ?x - q * ?y in if ?y ≤ r then (q + 1, r - ?y) else (q, r))` computes both with division only.

code-printing

constant `uint32-div ↪`
`(SML) Word32.div ((-), (-)) and`
`(Haskell) Prelude.div`
| constant `uint32-mod ↪`
`(SML) Word32.mod ((-), (-)) and`
`(Haskell) Prelude.mod`
| constant `uint32-divmod ↪`
`(Haskell) divmod`
| constant `uint32-sdiv ↪`
`(OCaml) Int32.div and`
`(Scala) - '/' -`

definition `uint32-test-bit :: uint32 ⇒ integer ⇒ bool`
where `[code del]:`
`uint32-test-bit x n =`
`(if n < 0 ∨ 31 < n then undefined (bit :: uint32 ⇒ -) x n`
`else bit x (nat-of-integer n))`

lemma `test-bit-uint32-code [code]:`
`bit x n ⟷ n < 32 ∧ uint32-test-bit x (integer-of-nat n)`

including *undefined-transfer integer.lifting* \langle proof \rangle

lemma *uint32-test-bit-code* [code]:

uint32-test-bit w n =
 (if $n < 0 \vee 31 < n$ then *undefined* (*bit* :: *uint32* \Rightarrow -) w n else *bit* (*Rep-uint32*
 w) (*nat-of-integer* n))
 \langle proof \rangle

code-printing constant *uint32-test-bit* \rightarrow

(SML) *Uint32.test'-bit* **and**
 (Haskell) *Data'-Bits.testBitBounded* **and**
 (OCaml) *Uint32.test'-bit* **and**
 (Scala) *Uint32.test'-bit* **and**
 (Eval) (fn $w \Rightarrow fn$ $n \Rightarrow$ if $n < 0$ orelse $32 \leq n$ then raise (*Fail* argument to
uint32'-test'-bit out of bounds) else *Uint32.test'-bit* w n)

definition *uint32-set-bit* :: *uint32* \Rightarrow *integer* \Rightarrow *bool* \Rightarrow *uint32*

where [code del]:

uint32-set-bit x n b =
 (if $n < 0 \vee 31 < n$ then *undefined* (*set-bit* :: *uint32* \Rightarrow -) x n b
 else *set-bit* x (*nat-of-integer* n) b)

lemma *set-bit-uint32-code* [code]:

set-bit x n b = (if $n < 32$ then *uint32-set-bit* x (*integer-of-nat* n) b else x)

including *undefined-transfer integer.lifting* \langle proof \rangle

lemma *uint32-set-bit-code* [code]:

Rep-uint32 (*uint32-set-bit* w n b) =
 (if $n < 0 \vee 31 < n$ then *Rep-uint32* (*undefined* (*set-bit* :: *uint32* \Rightarrow -) w n b)
 else *set-bit* (*Rep-uint32* w) (*nat-of-integer* n) b)

including *undefined-transfer* \langle proof \rangle

code-printing constant *uint32-set-bit* \rightarrow

(SML) *Uint32.set'-bit* **and**
 (Haskell) *Data'-Bits.setBitBounded* **and**
 (OCaml) *Uint32.set'-bit* **and**
 (Scala) *Uint32.set'-bit* **and**
 (Eval) (fn $w \Rightarrow fn$ $n \Rightarrow fn$ $b \Rightarrow$ if $n < 0$ orelse $32 \leq n$ then raise (*Fail*
 argument to *uint32'-set'-bit* out of bounds) else *Uint32.set'-bit* x n b)

definition *uint32-shiffl* :: *uint32* \Rightarrow *integer* \Rightarrow *uint32*

where [code del]:

uint32-shiffl x n = (if $n < 0 \vee 32 \leq n$ then *undefined* (*push-bit* :: *nat* \Rightarrow *uint32*
 \Rightarrow -) x n else *push-bit* (*nat-of-integer* n) x)

lemma *shiffl-uint32-code* [code]: *push-bit* n x = (if $n < 32$ then *uint32-shiffl* x
 (*integer-of-nat* n) else 0)

including *undefined-transfer integer.lifting* \langle proof \rangle

lemma *uint32-shiffl-code* [code]:

Rep-uint32 (uint32-shiffl w n) =
(if n < 0 ∨ 32 ≤ n then Rep-uint32 (undefined (push-bit :: nat ⇒ uint32 ⇒ -)
w n) else push-bit (nat-of-integer n) (Rep-uint32 w))
including *undefined-transfer* ⟨proof⟩

code-printing constant *uint32-shiffl* \rightarrow

(SML) Uint32.shiffl **and**
(Haskell) Data'-Bits.shifflBounded **and**
(OCaml) Uint32.shiffl **and**
(Scala) Uint32.shiffl **and**
(Eval) (fn x => fn i => if i < 0 orelse i ≥ 32 then raise Fail argument to
uint32'-shiffl out of bounds else Uint32.shiffl x i)

definition *uint32-shiftr* :: *uint32 ⇒ integer ⇒ uint32*

where [code del]:

uint32-shiftr x n = (if n < 0 ∨ 32 ≤ n then undefined (drop-bit :: nat ⇒ uint32
 \Rightarrow -) x n else drop-bit (nat-of-integer n) x)

lemma *shiftr-uint32-code* [code]: *drop-bit n x = (if n < 32 then uint32-shiftr x*
(integer-of-nat n) else 0)

including *undefined-transfer integer.lifting* ⟨proof⟩

lemma *uint32-shiftr-code* [code]:

Rep-uint32 (uint32-shiftr w n) =
(if n < 0 ∨ 32 ≤ n then Rep-uint32 (undefined (drop-bit :: nat ⇒ uint32 ⇒ -)
w n) else drop-bit (nat-of-integer n) (Rep-uint32 w))
including *undefined-transfer* ⟨proof⟩

code-printing constant *uint32-shiftr* \rightarrow

(SML) Uint32.shiftr **and**
(Haskell) Data'-Bits.shiftrBounded **and**
(OCaml) Uint32.shiftr **and**
(Scala) Uint32.shiftr **and**
(Eval) (fn x => fn i => if i < 0 orelse i ≥ 32 then raise Fail argument to
uint32'-shiftr out of bounds else Uint32.shiftr x i)

definition *uint32-sshiftr* :: *uint32 ⇒ integer ⇒ uint32*

where [code del]:

uint32-sshiftr x n =
(if n < 0 ∨ 32 ≤ n then undefined signed-drop-bit-uint32 n x else signed-drop-bit-uint32
(nat-of-integer n) x)

lemma *sshiftr-uint32-code* [code]:

signed-drop-bit-uint32 n x =
(if n < 32 then uint32-sshiftr x (integer-of-nat n) else if bit x 31 then - 1 else 0)
including *undefined-transfer integer.lifting* ⟨proof⟩

lemma *uint32-sshiftr-code* [code]:

```

Rep-uint32 (uint32-sshiftr w n) =
  (if n < 0 ∨ 32 ≤ n then Rep-uint32 (undefined signed-drop-bit-uint32 n w) else
  signed-drop-bit (nat-of-integer n) (Rep-uint32 w))
including undefined-transfer ⟨proof⟩

```

```

code-printing constant uint32-sshiftr →
  (SML) Uint32.shiftr'-signed and
  (Haskell)
  (Prelude.fromInteger (Prelude.toInteger (Data'-Bits.shiftrBounded (Prelude.fromInteger
  (Prelude.toInteger -) :: Uint32.Int32) -)) :: Uint32.Word32) and
  (OCaml) Uint32.shiftr'-signed and
  (Scala) Uint32.shiftr'-signed and
  (Eval) (fn x => fn i => if i < 0 orelse i ≥ 32 then raise Fail argument to
  uint32'-shiftr'-signed out of bounds else Uint32.shiftr'-signed x i)

```

context

```

includes bit-operations-syntax
begin

```

```

lemma uint32-msb-test-bit: msb x ↔ bit (x :: uint32) 31
  ⟨proof⟩

```

```

lemma msb-uint32-code [code]: msb x ↔ uint32-test-bit x 31
  ⟨proof⟩

```

```

lemma uint32-of-int-code [code]:
  uint32-of-int i = Uint32 (integer-of-int i)
  including integer.lifting ⟨proof⟩

```

```

lemma int-of-uint32-code [code]:
  int-of-uint32 x = int-of-integer (integer-of-uint32 x)
  including integer.lifting ⟨proof⟩

```

```

lemma uint32-of-nat-code [code]:
  uint32-of-nat = uint32-of-int ∘ int
  ⟨proof⟩

```

```

lemma nat-of-uint32-code [code]:
  nat-of-uint32 x = nat-of-integer (integer-of-uint32 x)
  ⟨proof⟩ including integer.lifting ⟨proof⟩

```

definition integer-of-uint32-signed :: uint32 ⇒ integer
where

```

integer-of-uint32-signed n = (if bit n 31 then undefined integer-of-uint32 n else
integer-of-uint32 n)

```

```

lemma integer-of-uint32-signed-code [code]:
  integer-of-uint32-signed n =
  (if bit n 31 then undefined integer-of-uint32 n else integer-of-int (uint (Rep-uint32'

```

```

n)))
  ⟨proof⟩

lemma integer-of-uint32-code [code]:
  integer-of-uint32 n =
    (if bit n 31 then integer-of-uint32-signed (n AND 0x7FFFFFFF) OR 0x80000000
    else integer-of-uint32-signed n)
  ⟨proof⟩
  including integer.lifting ⟨proof⟩

end

code-printing
  constant integer-of-uint32 ↪
    (SML) IntInf.fromLarge (Word32.toLargeInt -) : IntInf.int and
    (Haskell) Prelude.toInteger
| constant integer-of-uint32-signed ↪
    (OCaml) Z.of'-int32 and
    (Scala) BigInt

```

6.3 Quickcheck setup

```

definition uint32-of-natural :: natural ⇒ uint32
where uint32-of-natural x ≡ Uint32 (integer-of-natural x)

instantiation uint32 :: {random, exhaustive, full-exhaustive} begin
definition random-uint32 ≡ qc-random-cnv uint32-of-natural
definition exhaustive-uint32 ≡ qc-exhaustive-cnv uint32-of-natural
definition full-exhaustive-uint32 ≡ qc-full-exhaustive-cnv uint32-of-natural
instance ⟨proof⟩
end

instantiation uint32 :: narrowing begin

interpretation quickcheck-narrowing-samples
  λi. let x = Uint32 i in (x, 0xFFFFFFFF - x) 0
  Typerep.Typerep (STR "Uint32.uint32") [] ⟨proof⟩

definition narrowing-uint32 d = qc-narrowing-drawn-from (narrowing-samples d)
  d
declare [[code drop: partial-term-of :: uint32 itself ⇒ -]]
lemmas partial-term-of-uint32 [code] = partial-term-of-code

instance ⟨proof⟩
end

end

```

Chapter 7

Unsigned words of 16 bits

```
theory Uint16 imports  
  Word-Type-Copies  
  Code-Target-Integer-Bit  
begin
```

Restriction for ML code generation: This theory assumes that the ML system provides a `Word16` implementation (mlton does, but PolyML 5.5 does not). Therefore, the code setup lives in the target *SML-word* rather than *SML*. This ensures that code generation still works as long as *uint16* is not involved. For the target *SML* itself, no special code generation for this type is set up. Nevertheless, it should work by emulation via *16 word* if the theory *Code-Target-Int-Bit* is imported.

Restriction for OCaml code generation: OCaml does not provide an `int16` type, so no special code generation for this type is set up.

7.1 Type definition and primitive operations

```
typedef uint16 =  $\langle UNIV :: 16 \text{ word set} \rangle$   $\langle proof \rangle$   
  
global-interpretation uint16: word-type-copy Abs-uint16 Rep-uint16  
   $\langle proof \rangle$   
  
setup-lifting type-definition-uint16  
  
declare uint16.of-word-of [code abstype]  
  
declare Quotient-uint16 [transfer-rule]  
  
instantiation uint16 ::  $\langle \{ comm\text{-}ring\text{-}1, semiring\text{-}modulo, equal, linorder \} \rangle$   
begin  
  
lift-definition zero-uint16 :: uint16 is 0  $\langle proof \rangle$   
lift-definition one-uint16 :: uint16 is 1  $\langle proof \rangle$ 
```

```

lift-definition plus-uint16 :: ⟨uint16 ⇒ uint16 ⇒ uint16⟩ is ⟨(+)⟩ ⟨proof⟩
lift-definition uminus-uint16 :: ⟨uint16 ⇒ uint16⟩ is uminus ⟨proof⟩
lift-definition minus-uint16 :: ⟨uint16 ⇒ uint16 ⇒ uint16⟩ is ⟨(-)⟩ ⟨proof⟩
lift-definition times-uint16 :: ⟨uint16 ⇒ uint16 ⇒ uint16⟩ is ⟨(*)⟩ ⟨proof⟩
lift-definition divide-uint16 :: ⟨uint16 ⇒ uint16 ⇒ uint16⟩ is ⟨(div)⟩ ⟨proof⟩
lift-definition modulo-uint16 :: ⟨uint16 ⇒ uint16 ⇒ uint16⟩ is ⟨(mod)⟩ ⟨proof⟩
lift-definition equal-uint16 :: ⟨uint16 ⇒ uint16 ⇒ bool⟩ is ⟨HOL.equal⟩ ⟨proof⟩
lift-definition less-eq-uint16 :: ⟨uint16 ⇒ uint16 ⇒ bool⟩ is ⟨(≤)⟩ ⟨proof⟩
lift-definition less-uint16 :: ⟨uint16 ⇒ uint16 ⇒ bool⟩ is ⟨(<)⟩ ⟨proof⟩

```

```

global-interpretation uint16: word-type-copy-ring Abs-uint16 Rep-uint16
  ⟨proof⟩

```

```

instance ⟨proof⟩

```

```

end

```

```

instantiation uint16 :: ring-bit-operations
begin

```

```

lift-definition bit-uint16 :: ⟨uint16 ⇒ nat ⇒ bool⟩ is bit ⟨proof⟩
lift-definition not-uint16 :: ⟨uint16 ⇒ uint16⟩ is ⟨Bit-Operations.not⟩ ⟨proof⟩
lift-definition and-uint16 :: ⟨uint16 ⇒ uint16 ⇒ uint16⟩ is ⟨Bit-Operations.and⟩
  ⟨proof⟩
lift-definition or-uint16 :: ⟨uint16 ⇒ uint16 ⇒ uint16⟩ is ⟨Bit-Operations.or⟩
  ⟨proof⟩
lift-definition xor-uint16 :: ⟨uint16 ⇒ uint16 ⇒ uint16⟩ is ⟨Bit-Operations.xor⟩
  ⟨proof⟩
lift-definition mask-uint16 :: ⟨nat ⇒ uint16⟩ is mask ⟨proof⟩
lift-definition push-bit-uint16 :: ⟨nat ⇒ uint16 ⇒ uint16⟩ is push-bit ⟨proof⟩
lift-definition drop-bit-uint16 :: ⟨nat ⇒ uint16 ⇒ uint16⟩ is drop-bit ⟨proof⟩
lift-definition signed-drop-bit-uint16 :: ⟨nat ⇒ uint16 ⇒ uint16⟩ is signed-drop-bit
  ⟨proof⟩
lift-definition take-bit-uint16 :: ⟨nat ⇒ uint16 ⇒ uint16⟩ is take-bit ⟨proof⟩
lift-definition set-bit-uint16 :: ⟨nat ⇒ uint16 ⇒ uint16⟩ is Bit-Operations.set-bit
  ⟨proof⟩
lift-definition unset-bit-uint16 :: ⟨nat ⇒ uint16 ⇒ uint16⟩ is unset-bit ⟨proof⟩
lift-definition flip-bit-uint16 :: ⟨nat ⇒ uint16 ⇒ uint16⟩ is flip-bit ⟨proof⟩

```

```

global-interpretation uint16: word-type-copy-bits Abs-uint16 Rep-uint16 signed-drop-bit-uint16
  ⟨proof⟩

```

```

instance
  ⟨proof⟩

```

```

end

```

```

lift-definition uint16-of-nat :: ⟨nat ⇒ uint16⟩
  is word-of-nat ⟨proof⟩

```


lift-definition *nat-of-uint16* :: $\langle \text{uint16} \Rightarrow \text{nat} \rangle$
is *unat* $\langle \text{proof} \rangle$

lift-definition *uint16-of-int* :: $\langle \text{int} \Rightarrow \text{uint16} \rangle$
is *word-of-int* $\langle \text{proof} \rangle$

lift-definition *int-of-uint16* :: $\langle \text{uint16} \Rightarrow \text{int} \rangle$
is *uint* $\langle \text{proof} \rangle$

context
includes *integer.lifting*
begin

lift-definition *Uint16* :: $\langle \text{integer} \Rightarrow \text{uint16} \rangle$
is *word-of-int* $\langle \text{proof} \rangle$

lift-definition *integer-of-uint16* :: $\langle \text{uint16} \Rightarrow \text{integer} \rangle$
is *uint* $\langle \text{proof} \rangle$

end

global-interpretation *uint16*: *word-type-copy-more Abs-uint16 Rep-uint16 signed-drop-bit-uint16*
uint16-of-nat nat-of-uint16 uint16-of-int int-of-uint16 Uint16 integer-of-uint16
 $\langle \text{proof} \rangle$

instantiation *uint16* :: $\{ \text{size}, \text{msb}, \text{lsb}, \text{set-bit}, \text{bit-comprehension} \}$
begin

lift-definition *size-uint16* :: $\langle \text{uint16} \Rightarrow \text{nat} \rangle$ **is** *size* $\langle \text{proof} \rangle$

lift-definition *msb-uint16* :: $\langle \text{uint16} \Rightarrow \text{bool} \rangle$ **is** *msb* $\langle \text{proof} \rangle$

lift-definition *lsb-uint16* :: $\langle \text{uint16} \Rightarrow \text{bool} \rangle$ **is** *lsb* $\langle \text{proof} \rangle$

Workaround: avoid name space clash by spelling out *lift-definition* explicitly.

definition *set-bit-uint16* :: $\langle \text{uint16} \Rightarrow \text{nat} \Rightarrow \text{bool} \Rightarrow \text{uint16} \rangle$
where *set-bit-uint16-eq*: $\langle \text{set-bit-uint16 } a \ n \ b = (\text{if } b \ \text{then } \text{Bit-Operations.set-bit} \ \text{else } \text{unset-bit}) \ n \ a \rangle$

context
includes *lifting-syntax*
begin

lemma *set-bit-uint16-transfer* [*transfer-rule*]:
 $\langle (\text{cr-uint16} \ \text{====} \ (=) \ \text{====} \ (\longleftrightarrow) \ \text{====} \ \text{cr-uint16}) \ \text{Generic-set-bit.set-bit} \ \text{Generic-set-bit.set-bit} \rangle$
 $\langle \text{proof} \rangle$

end

lift-definition *set-bits-uint16* :: $\langle (nat \Rightarrow bool) \Rightarrow uint16 \rangle$ **is** *set-bits* $\langle proof \rangle$
lift-definition *set-bits-aux-uint16* :: $\langle (nat \Rightarrow bool) \Rightarrow nat \Rightarrow uint16 \Rightarrow uint16 \rangle$ **is**
set-bits-aux $\langle proof \rangle$

global-interpretation *uint16*: *word-type-copy-misc Abs-uint16 Rep-uint16 signed-drop-bit-uint16*
uint16-of-nat nat-of-uint16 uint16-of-int int-of-uint16 Uint16 integer-of-uint16 16
set-bits-aux-uint16
 $\langle proof \rangle$

instance $\langle proof \rangle$

end

7.2 Code setup

code-printing code-module *Uint16* \rightarrow (*SML-word*)
 $\langle (*$ Test that words can handle numbers between 0 and 15 *)
val - = *if* 4 <= *Word.wordSize* *then* () *else raise* (*Fail* (*wordSize* *less than* 4));

structure *Uint16* : *sig*
val *set-bit* : *Word16.word* \rightarrow *IntInf.int* \rightarrow *bool* \rightarrow *Word16.word*
val *shiffl* : *Word16.word* \rightarrow *IntInf.int* \rightarrow *Word16.word*
val *shiftr* : *Word16.word* \rightarrow *IntInf.int* \rightarrow *Word16.word*
val *shiftr-signed* : *Word16.word* \rightarrow *IntInf.int* \rightarrow *Word16.word*
val *test-bit* : *Word16.word* \rightarrow *IntInf.int* \rightarrow *bool*
end = *struct*

fun *set-bit* *x n b* =
let *val* *mask* = *Word16.<<* (*0wx1*, *Word.fromLargeInt* (*IntInf.toLarge* *n*))
in *if* *b* *then* *Word16.orb* (*x*, *mask*)
else *Word16.andb* (*x*, *Word16.notb* *mask*)
end

fun *shiffl* *x n* =
Word16.<< (*x*, *Word.fromLargeInt* (*IntInf.toLarge* *n*))

fun *shiftr* *x n* =
Word16.>> (*x*, *Word.fromLargeInt* (*IntInf.toLarge* *n*))

fun *shiftr-signed* *x n* =
Word16.~>> (*x*, *Word.fromLargeInt* (*IntInf.toLarge* *n*))

fun *test-bit* *x n* =
Word16.andb (*x*, *Word16.<<* (*0wx1*, *Word.fromLargeInt* (*IntInf.toLarge* *n*)))
 $\langle >$ *Word16.fromInt* 0

end; (* *struct* *Uint16* *)
code-reserved *SML-word* *Uint16*

code-printing code-module *Uint16* \rightarrow (*Haskell*)
 \langle module *Uint16*(*Int16*, *Word16*) where

import Data.Int(Int16)
import Data.Word(Word16)
code-reserved *Haskell Uint16*

Scala provides unsigned 16-bit numbers as Char.

code-printing code-module *Uint16* \rightarrow (*Scala*)
 \langle object *Uint16* {

def set-bit(*x*: *scala.Char*, *n*: *BigInt*, *b*: *Boolean*) : *scala.Char* =
b match {
 case true => (*x* | (*1.toChar* << *n.intValue*)).*toChar*
 case false => (*x* & (*1.toChar* << *n.intValue*).*unary~*).*toChar*
}

def shiftl(*x*: *scala.Char*, *n*: *BigInt*) : *scala.Char* = (*x* << *n.intValue*).*toChar*

def shiftr(*x*: *scala.Char*, *n*: *BigInt*) : *scala.Char* = (*x* >>> *n.intValue*).*toChar*

def shiftr-signed(*x*: *scala.Char*, *n*: *BigInt*) : *scala.Char* = (*x.toShort* >> *n.intValue*).*toChar*

def test-bit(*x*: *scala.Char*, *n*: *BigInt*) : *Boolean* = (*x* & (*1.toChar* << *n.intValue*))
!= 0

} /* object *Uint16* */

code-reserved *Scala Uint16*

Avoid *Abs-uint16* in generated code, use *Rep-uint16'* instead. The symbolic implementations for *code_simp* use *Rep-uint16*.

The new destructor *Rep-uint16'* is executable. As the simplifier is given the [code abstract] equations literally, we cannot implement *Rep-uint16* directly, because that makes *code_simp* loop.

If code generation raises *Match*, some equation probably contains *Rep-uint16* ([code abstract] equations for *uint16* may use *Rep-uint16* because these instances will be folded away.)

To convert 16 word values into *uint16*, use *Abs-uint16'*.

definition *Rep-uint16'* where [*simp*]: *Rep-uint16'* = *Rep-uint16*

lemma *Rep-uint16'-transfer* [*transfer-rule*]:

rel-fun cr-uint16 (=) ($\lambda x. x$) *Rep-uint16'*
 \langle proof \rangle

lemma *Rep-uint16'-code* [*code*]: *Rep-uint16'* *x* = (*BITS n. bit x n*)

\langle proof \rangle

lift-definition *Abs-uint16'* :: 16 word \Rightarrow uint16 is $\lambda x :: 16 \text{ word. } x$ *<proof>*

lemma *Abs-uint16'-code* [code]:

Abs-uint16' x = Uint16 (integer-of-int (uint x))

including *integer.lifting* *<proof>*

declare [[code drop: term-of-class.term-of :: uint16 \Rightarrow -]]

lemma *term-of-uint16-code* [code]:

defines *TR* \equiv *typerep.TypeRep* **and** *bit0* \equiv *STR "Numeral-Type.bit0"* **shows**
term-of-class.term-of x =

Code-Evaluation.App (Code-Evaluation.Const (STR "Uint16.uint16.Abs-uint16")
(TR (STR "fun") [TR (STR "Word.word") [TR bit0 [TR bit0 [TR bit0 [TR bit0
[TR (STR "Numeral-Type.num1") []]]], TR (STR "Uint16.uint16") []])
(term-of-class.term-of (Rep-uint16' x))
<proof>

lemma *Uint16-code* [code]: *Rep-uint16 (Uint16 i) = word-of-int (int-of-integer-symbolic i)*

<proof>

code-printing

type-constructor *uint16* \rightarrow

(*SML-word*) *Word16.word* **and**

(*Haskell*) *Uint16.Word16* **and**

(*Scala*) *scala.Char*

| **constant** *Uint16* \rightarrow

(*SML-word*) *Word16.fromLargeInt (IntInf.toLarge -)* **and**

(*Haskell*) (*Prelude.fromInteger - :: Uint16.Word16*) **and**

(*Haskell-Quickcheck*) (*Prelude.fromInteger (Prelude.toInteger -) :: Uint16.Word16*)

and

(*Scala*) *-.charValue*

| **constant** *0* :: *uint16* \rightarrow

(*SML-word*) (*Word16.fromInt 0*) **and**

(*Haskell*) (*0 :: Uint16.Word16*) **and**

(*Scala*) *0*

| **constant** *1* :: *uint16* \rightarrow

(*SML-word*) (*Word16.fromInt 1*) **and**

(*Haskell*) (*1 :: Uint16.Word16*) **and**

(*Scala*) *1*

| **constant** *plus* :: *uint16* \Rightarrow - \Rightarrow - \rightarrow

(*SML-word*) *Word16.+ ((-), (-))* **and**

(*Haskell*) **infixl 6 +** **and**

(*Scala*) (*- +/ -*).*toChar*

| **constant** *uminus* :: *uint16* \Rightarrow - \rightarrow

(*SML-word*) *Word16.~* **and**

(*Haskell*) **negate** **and**

(*Scala*) (*- -*).*toChar*

```

| constant minus :: uint16 ⇒ - →
  (SML-word) Word16.- ((-), (-)) and
  (Haskell) infixl 6 - and
  (Scala) (- - / -).toChar
| constant times :: uint16 ⇒ - ⇒ - →
  (SML-word) Word16.* ((-), (-)) and
  (Haskell) infixl 7 * and
  (Scala) (- * / -).toChar
| constant HOL.equal :: uint16 ⇒ - ⇒ bool →
  (SML-word) !((- : Word16.word) = -) and
  (Haskell) infix 4 == and
  (Scala) infixl 5 ==
| class-instance uint16 :: equal → (Haskell) -
| constant less-eq :: uint16 ⇒ - ⇒ bool →
  (SML-word) Word16.<= ((-), (-)) and
  (Haskell) infix 4 <= and
  (Scala) infixl 4 <=
| constant less :: uint16 ⇒ - ⇒ bool →
  (SML-word) Word16.< ((-), (-)) and
  (Haskell) infix 4 < and
  (Scala) infixl 4 <
| constant Bit-Operations.not :: uint16 ⇒ - →
  (SML-word) Word16.notb and
  (Haskell) Data'-Bits.complement and
  (Scala) -.unary'~.toChar
| constant Bit-Operations.and :: uint16 ⇒ - →
  (SML-word) Word16.andb ((-), / (-)) and
  (Haskell) infixl 7 Data-Bits.&. and
  (Scala) (- & -).toChar
| constant Bit-Operations.or :: uint16 ⇒ - →
  (SML-word) Word16.orb ((-), / (-)) and
  (Haskell) infixl 5 Data-Bits.|. and
  (Scala) (- | -).toChar
| constant Bit-Operations.xor :: uint16 ⇒ - →
  (SML-word) Word16.xorb ((-), / (-)) and
  (Haskell) Data'-Bits.xor and
  (Scala) (- ^ -).toChar

definition uint16-div :: uint16 ⇒ uint16 ⇒ uint16
where uint16-div x y = (if y = 0 then undefined ((div) :: uint16 ⇒ -) x (0 :: uint16) else x div y)

definition uint16-mod :: uint16 ⇒ uint16 ⇒ uint16
where uint16-mod x y = (if y = 0 then undefined ((mod) :: uint16 ⇒ -) x (0 :: uint16) else x mod y)

context includes undefined-transfer begin

lemma div-uint16-code [code]: x div y = (if y = 0 then 0 else uint16-div x y)

```

<proof>

lemma *mod-uint16-code* [code]: $x \text{ mod } y = (\text{if } y = 0 \text{ then } x \text{ else } \text{uint16-mod } x \ y)$
<proof>

lemma *uint16-div-code* [code]:
 $\text{Rep-uint16 } (\text{uint16-div } x \ y) =$
 $(\text{if } y = 0 \text{ then } \text{Rep-uint16 } (\text{undefined } ((\text{div}) :: \text{uint16} \Rightarrow -) \ x \ (0 :: \text{uint16})) \text{ else } \text{Rep-uint16 } x \ \text{div } \text{Rep-uint16 } y)$
<proof>

lemma *uint16-mod-code* [code]:
 $\text{Rep-uint16 } (\text{uint16-mod } x \ y) =$
 $(\text{if } y = 0 \text{ then } \text{Rep-uint16 } (\text{undefined } ((\text{mod}) :: \text{uint16} \Rightarrow -) \ x \ (0 :: \text{uint16})) \text{ else } \text{Rep-uint16 } x \ \text{mod } \text{Rep-uint16 } y)$
<proof>

end

code-printing constant *uint16-div* \rightarrow
(SML-word) *Word16.div* ((-), (-)) **and**
(Haskell) *Prelude.div* **and**
(Scala) (- '/' -).toChar
| **constant** *uint16-mod* \rightarrow
(SML-word) *Word16.mod* ((-), (-)) **and**
(Haskell) *Prelude.mod* **and**
(Scala) (- '%' -).toChar

definition *uint16-test-bit* :: $\text{uint16} \Rightarrow \text{integer} \Rightarrow \text{bool}$
where [code del]:
 $\text{uint16-test-bit } x \ n =$
 $(\text{if } n < 0 \vee 15 < n \text{ then } \text{undefined } (\text{bit} :: \text{uint16} \Rightarrow -) \ x \ n$
 $\text{else } \text{bit } x \ (\text{nat-of-integer } n))$

lemma *test-bit-uint16-code* [code]:
 $\text{bit } x \ n \longleftrightarrow n < 16 \wedge \text{uint16-test-bit } x \ (\text{integer-of-nat } n)$
including *undefined-transfer integer.lifting* *<proof>*

lemma *uint16-test-bit-code* [code]:
 $\text{uint16-test-bit } w \ n =$
 $(\text{if } n < 0 \vee 15 < n \text{ then } \text{undefined } (\text{bit} :: \text{uint16} \Rightarrow -) \ w \ n \text{ else } \text{bit } (\text{Rep-uint16 } w) \ (\text{nat-of-integer } n))$
<proof>

code-printing constant *uint16-test-bit* \rightarrow
(SML-word) *Uint16.test'-bit* **and**
(Haskell) *Data'-Bits.testBitBounded* **and**
(Scala) *Uint16.test'-bit*

definition *uint16-set-bit* :: *uint16* \Rightarrow *integer* \Rightarrow *bool* \Rightarrow *uint16*

where [code del]:

uint16-set-bit *x n b* =
 (if $n < 0 \vee 15 < n$ then undefined (*set-bit* :: *uint16* \Rightarrow -) *x n b*
 else *set-bit* *x* (*nat-of-integer* *n*) *b*)

lemma *set-bit-uint16-code* [code]:

set-bit *x n b* = (if $n < 16$ then *uint16-set-bit* *x* (*integer-of-nat* *n*) *b* else *x*)

including *undefined-transfer integer.lifting* <proof>

lemma *uint16-set-bit-code* [code]:

Rep-uint16 (*uint16-set-bit* *w n b*) =
 (if $n < 0 \vee 15 < n$ then *Rep-uint16* (undefined (*set-bit* :: *uint16* \Rightarrow -) *w n b*)
 else *set-bit* (*Rep-uint16* *w*) (*nat-of-integer* *n*) *b*)

including *undefined-transfer* <proof>

code-printing constant *uint16-set-bit* \rightarrow

(SML-word) *Uint16.set'-bit* **and**
 (Haskell) *Data'-Bits.setBitBounded* **and**
 (Scala) *Uint16.set'-bit*

definition *uint16-shiffl* :: *uint16* \Rightarrow *integer* \Rightarrow *uint16*

where [code del]:

uint16-shiffl *x n* = (if $n < 0 \vee 16 \leq n$ then undefined (*push-bit* :: *nat* \Rightarrow *uint16* \Rightarrow -) *x n* else *push-bit* (*nat-of-integer* *n*) *x*)

lemma *shiffl-uint16-code* [code]: *push-bit* *n x* = (if $n < 16$ then *uint16-shiffl* *x* (*integer-of-nat* *n*) else 0)

including *undefined-transfer integer.lifting* <proof>

lemma *uint16-shiffl-code* [code]:

Rep-uint16 (*uint16-shiffl* *w n*) =
 (if $n < 0 \vee 16 \leq n$ then *Rep-uint16* (undefined (*push-bit* :: *nat* \Rightarrow *uint16* \Rightarrow -) *w n*)
 else *push-bit* (*nat-of-integer* *n*) (*Rep-uint16* *w*))

including *undefined-transfer* <proof>

code-printing constant *uint16-shiffl* \rightarrow

(SML-word) *Uint16.shiffl* **and**
 (Haskell) *Data'-Bits.shifflBounded* **and**
 (Scala) *Uint16.shiffl*

definition *uint16-shiftr* :: *uint16* \Rightarrow *integer* \Rightarrow *uint16*

where [code del]:

uint16-shiftr *x n* = (if $n < 0 \vee 16 \leq n$ then undefined (*drop-bit* :: *nat* \Rightarrow *uint16* \Rightarrow -) *x n* else *drop-bit* (*nat-of-integer* *n*) *x*)

lemma *shiftr-uint16-code* [code]: *drop-bit* *n x* = (if $n < 16$ then *uint16-shiftr* *x* (*integer-of-nat* *n*) else 0)

including *undefined-transfer integer.lifting* \langle proof \rangle

lemma *uint16-shiftr-code* [code]:

Rep-uint16 (uint16-shiftr w n) =
(if n < 0 \vee 16 \leq n then Rep-uint16 (undefined (drop-bit :: nat \Rightarrow uint16 \Rightarrow -)
w n)
else drop-bit (nat-of-integer n) (Rep-uint16 w))

including *undefined-transfer* \langle proof \rangle

code-printing constant *uint16-shiftr* \rightarrow

(SML-word) Uint16.shiftr **and**
(Haskell) Data'-Bits.shiftrBounded **and**
(Scala) Uint16.shiftr

definition *uint16-sshiftr* :: *uint16 \Rightarrow integer \Rightarrow uint16*

where [code del]:

uint16-sshiftr x n =
(if n < 0 \vee 16 \leq n then undefined signed-drop-bit-uint16 n x else signed-drop-bit-uint16
(nat-of-integer n) x)

lemma *sshiftr-uint16-code* [code]:

signed-drop-bit-uint16 n x =
(if n < 16 then uint16-sshiftr x (integer-of-nat n) else if bit x 15 then -1 else 0)
including *undefined-transfer integer.lifting* \langle proof \rangle

lemma *uint16-sshiftr-code* [code]:

Rep-uint16 (uint16-sshiftr w n) =
(if n < 0 \vee 16 \leq n then Rep-uint16 (undefined signed-drop-bit-uint16 n w)
else signed-drop-bit (nat-of-integer n) (Rep-uint16 w))
including *undefined-transfer* \langle proof \rangle

code-printing constant *uint16-sshiftr* \rightarrow

(SML-word) Uint16.shiftr'-signed **and**
(Haskell)
(Prelude.fromInteger (Prelude.toInteger (Data'-Bits.shiftrBounded (Prelude.fromInteger
(Prelude.toInteger -) :: Uint16.Int16) -)) :: Uint16.Word16) **and**
(Scala) Uint16.shiftr'-signed

lemma *uint16-msb-test-bit*: *msb x \longleftrightarrow bit (x :: uint16) 15*

\langle proof \rangle

lemma *msb-uint16-code* [code]: *msb x \longleftrightarrow uint16-test-bit x 15*

\langle proof \rangle

lemma *uint16-of-int-code* [code]: *uint16-of-int i = Uint16 (integer-of-int i)*

including *integer.lifting* \langle proof \rangle

lemma *int-of-uint16-code* [code]:

int-of-uint16 x = int-of-integer (integer-of-uint16 x)

<proof>

lemma *uint16-of-nat-code* [code]:
uint16-of-nat = *uint16-of-int* \circ *int*
<proof>

lemma *nat-of-uint16-code* [code]:
nat-of-uint16 *x* = *nat-of-integer* (*integer-of-uint16* *x*)
<proof> **including** *integer.lifting* *<proof>*

lemma *integer-of-uint16-code* [code]:
integer-of-uint16 *n* = *integer-of-int* (*uint* (*Rep-uint16'* *n*))
<proof>

code-printing

constant *integer-of-uint16* \rightarrow
(SML-word) *Word16.toInt* - : *IntInf.int* **and**
(Haskell) *Prelude.toInteger* **and**
(Scala) *BigInt*

7.3 Quickcheck setup

definition *uint16-of-natural* :: *natural* \Rightarrow *uint16*
where *uint16-of-natural* *x* \equiv *Uint16* (*integer-of-natural* *x*)

instantiation *uint16* :: {*random, exhaustive, full-exhaustive*} **begin**

definition *random-uint16* \equiv *qc-random-cnv* *uint16-of-natural*

definition *exhaustive-uint16* \equiv *qc-exhaustive-cnv* *uint16-of-natural*

definition *full-exhaustive-uint16* \equiv *qc-full-exhaustive-cnv* *uint16-of-natural*

instance *<proof>*

end

instantiation *uint16* :: *narrowing* **begin**

interpretation *quickcheck-narrowing-samples*

$\lambda i.$ *let* *x* = *Uint16* *i* *in* (*x*, *0xFFFF* - *x*) 0

Typerep.Typerep (*STR* "*Uint16.uint16'*") [] *<proof>*

definition *narrowing-uint16* *d* = *qc-narrowing-drawn-from* (*narrowing-samples* *d*)

declare [[code drop: *partial-term-of* :: *uint16* *itself* \Rightarrow -]]

lemmas *partial-term-of-uint16* [code] = *partial-term-of-code*

instance *<proof>*

end

end

Chapter 8

Unsigned words of 8 bits

```
theory Uint8 imports  
  Word-Type-Copies  
  Code-Target-Integer-Bit  
begin
```

Restriction for OCaml code generation: OCaml does not provide an `int8` type, so no special code generation for this type is set up. If the theory *Code-Target-Int-Bit* is imported, the type *uint8* is emulated via *8 word*.

8.1 Type definition and primitive operations

```
typedef uint8 =  $\langle UNIV :: 8\ word\ set \rangle$   $\langle proof \rangle$   
  
global-interpretation uint8: word-type-copy Abs-uint8 Rep-uint8  
   $\langle proof \rangle$   
  
setup-lifting type-definition-uint8  
  
declare uint8.of-word-of [code abstype]  
  
declare Quotient-uint8 [transfer-rule]  
  
instantiation uint8 ::  $\langle \{ comm-ring-1, semiring-modulo, equal, linorder \} \rangle$   
begin  
  
lift-definition zero-uint8 :: uint8 is 0  $\langle proof \rangle$   
lift-definition one-uint8 :: uint8 is 1  $\langle proof \rangle$   
lift-definition plus-uint8 ::  $\langle uint8 \Rightarrow uint8 \Rightarrow uint8 \rangle$  is  $\langle (+) \rangle$   $\langle proof \rangle$   
lift-definition uminus-uint8 ::  $\langle uint8 \Rightarrow uint8 \rangle$  is uminus  $\langle proof \rangle$   
lift-definition minus-uint8 ::  $\langle uint8 \Rightarrow uint8 \Rightarrow uint8 \rangle$  is  $\langle (-) \rangle$   $\langle proof \rangle$   
lift-definition times-uint8 ::  $\langle uint8 \Rightarrow uint8 \Rightarrow uint8 \rangle$  is  $\langle (*) \rangle$   $\langle proof \rangle$   
lift-definition divide-uint8 ::  $\langle uint8 \Rightarrow uint8 \Rightarrow uint8 \rangle$  is  $\langle (div) \rangle$   $\langle proof \rangle$   
lift-definition modulo-uint8 ::  $\langle uint8 \Rightarrow uint8 \Rightarrow uint8 \rangle$  is  $\langle (mod) \rangle$   $\langle proof \rangle$   
lift-definition equal-uint8 ::  $\langle uint8 \Rightarrow uint8 \Rightarrow bool \rangle$  is  $\langle HOL.equal \rangle$   $\langle proof \rangle$ 
```

lift-definition *less-eq-uint8* :: $\langle \text{uint8} \Rightarrow \text{uint8} \Rightarrow \text{bool} \rangle$ **is** $\langle (\leq) \rangle$ $\langle \text{proof} \rangle$

lift-definition *less-uint8* :: $\langle \text{uint8} \Rightarrow \text{uint8} \Rightarrow \text{bool} \rangle$ **is** $\langle (<) \rangle$ $\langle \text{proof} \rangle$

global-interpretation *uint8*: *word-type-copy-ring Abs-uint8 Rep-uint8*
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

instantiation *uint8* :: *ring-bit-operations*
begin

lift-definition *bit-uint8* :: $\langle \text{uint8} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$ **is** *bit* $\langle \text{proof} \rangle$

lift-definition *not-uint8* :: $\langle \text{uint8} \Rightarrow \text{uint8} \rangle$ **is** $\langle \text{Bit-Operations.not} \rangle$ $\langle \text{proof} \rangle$

lift-definition *and-uint8* :: $\langle \text{uint8} \Rightarrow \text{uint8} \Rightarrow \text{uint8} \rangle$ **is** $\langle \text{Bit-Operations.and} \rangle$
 $\langle \text{proof} \rangle$

lift-definition *or-uint8* :: $\langle \text{uint8} \Rightarrow \text{uint8} \Rightarrow \text{uint8} \rangle$ **is** $\langle \text{Bit-Operations.or} \rangle$ $\langle \text{proof} \rangle$

lift-definition *xor-uint8* :: $\langle \text{uint8} \Rightarrow \text{uint8} \Rightarrow \text{uint8} \rangle$ **is** $\langle \text{Bit-Operations.xor} \rangle$ $\langle \text{proof} \rangle$

lift-definition *mask-uint8* :: $\langle \text{nat} \Rightarrow \text{uint8} \rangle$ **is** *mask* $\langle \text{proof} \rangle$

lift-definition *push-bit-uint8* :: $\langle \text{nat} \Rightarrow \text{uint8} \Rightarrow \text{uint8} \rangle$ **is** *push-bit* $\langle \text{proof} \rangle$

lift-definition *drop-bit-uint8* :: $\langle \text{nat} \Rightarrow \text{uint8} \Rightarrow \text{uint8} \rangle$ **is** *drop-bit* $\langle \text{proof} \rangle$

lift-definition *signed-drop-bit-uint8* :: $\langle \text{nat} \Rightarrow \text{uint8} \Rightarrow \text{uint8} \rangle$ **is** *signed-drop-bit*
 $\langle \text{proof} \rangle$

lift-definition *take-bit-uint8* :: $\langle \text{nat} \Rightarrow \text{uint8} \Rightarrow \text{uint8} \rangle$ **is** *take-bit* $\langle \text{proof} \rangle$

lift-definition *set-bit-uint8* :: $\langle \text{nat} \Rightarrow \text{uint8} \Rightarrow \text{uint8} \rangle$ **is** $\langle \text{Bit-Operations.set-bit} \rangle$
 $\langle \text{proof} \rangle$

lift-definition *unset-bit-uint8* :: $\langle \text{nat} \Rightarrow \text{uint8} \Rightarrow \text{uint8} \rangle$ **is** *unset-bit* $\langle \text{proof} \rangle$

lift-definition *flip-bit-uint8* :: $\langle \text{nat} \Rightarrow \text{uint8} \Rightarrow \text{uint8} \rangle$ **is** *flip-bit* $\langle \text{proof} \rangle$

global-interpretation *uint8*: *word-type-copy-bits Abs-uint8 Rep-uint8 signed-drop-bit-uint8*
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

lift-definition *uint8-of-nat* :: $\langle \text{nat} \Rightarrow \text{uint8} \rangle$
is *word-of-nat* $\langle \text{proof} \rangle$

lift-definition *nat-of-uint8* :: $\langle \text{uint8} \Rightarrow \text{nat} \rangle$
is *unat* $\langle \text{proof} \rangle$

lift-definition *uint8-of-int* :: $\langle \text{int} \Rightarrow \text{uint8} \rangle$
is *word-of-int* $\langle \text{proof} \rangle$

lift-definition *int-of-uint8* :: $\langle \text{uint8} \Rightarrow \text{int} \rangle$
is *uint* $\langle \text{proof} \rangle$

```

context
  includes integer.lifting
begin

lift-definition Uint8 ::  $\langle \text{integer} \Rightarrow \text{uint8} \rangle$ 
  is word-of-int  $\langle \text{proof} \rangle$ 

lift-definition integer-of-uint8 ::  $\langle \text{uint8} \Rightarrow \text{integer} \rangle$ 
  is uint  $\langle \text{proof} \rangle$ 

end

global-interpretation uint8: word-type-copy-more Abs-uint8 Rep-uint8 signed-drop-bit-uint8
  uint8-of-nat nat-of-uint8 uint8-of-int int-of-uint8 Uint8 integer-of-uint8
   $\langle \text{proof} \rangle$ 

instantiation uint8 :: {size, msb, lsb, set-bit, bit-comprehension}
begin

lift-definition size-uint8 ::  $\langle \text{uint8} \Rightarrow \text{nat} \rangle$  is size  $\langle \text{proof} \rangle$ 

lift-definition msb-uint8 ::  $\langle \text{uint8} \Rightarrow \text{bool} \rangle$  is msb  $\langle \text{proof} \rangle$ 
lift-definition lsb-uint8 ::  $\langle \text{uint8} \Rightarrow \text{bool} \rangle$  is lsb  $\langle \text{proof} \rangle$ 

Workaround: avoid name space clash by spelling out lift-definition explicitly.

definition set-bit-uint8 ::  $\langle \text{uint8} \Rightarrow \text{nat} \Rightarrow \text{bool} \Rightarrow \text{uint8} \rangle$ 
  where set-bit-uint8-eq:  $\langle \text{set-bit-uint8 } a \ n \ b = (\text{if } b \ \text{then } \text{Bit-Operations.set-bit} \ \text{else } \text{unset-bit}) \ n \ a \rangle$ 

context
  includes lifting-syntax
begin

lemma set-bit-uint8-transfer [transfer-rule]:
   $\langle (\text{cr-uint8} \ ==\!>\!> \ (=) \ ==\!>\!> \ (\longleftrightarrow) \ ==\!>\!> \ \text{cr-uint8}) \ \text{Generic-set-bit.set-bit}$ 
  Generic-set-bit.set-bit
   $\langle \text{proof} \rangle$ 

end

lift-definition set-bits-uint8 ::  $\langle (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{uint8} \rangle$  is set-bits  $\langle \text{proof} \rangle$ 
lift-definition set-bits-aux-uint8 ::  $\langle (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow \text{uint8} \Rightarrow \text{uint8} \rangle$  is
set-bits-aux  $\langle \text{proof} \rangle$ 

global-interpretation uint8: word-type-copy-misc Abs-uint8 Rep-uint8 signed-drop-bit-uint8
  uint8-of-nat nat-of-uint8 uint8-of-int int-of-uint8 Uint8 integer-of-uint8 8 set-bits-aux-uint8
   $\langle \text{proof} \rangle$ 

```

```
instance <proof>
```

```
end
```

8.2 Code setup

```
code-printing code-module Uint8  $\rightarrow$  (SML)
```

```
<(* Test that words can handle numbers between 0 and 3 *)
```

```
val - = if 3 <= Word.wordSize then () else raise (Fail (wordSize less than 3));
```

```
structure Uint8 : sig
```

```
  val set-bit : Word8.word  $\rightarrow$  IntInf.int  $\rightarrow$  bool  $\rightarrow$  Word8.word
```

```
  val shiftl : Word8.word  $\rightarrow$  IntInf.int  $\rightarrow$  Word8.word
```

```
  val shiftr : Word8.word  $\rightarrow$  IntInf.int  $\rightarrow$  Word8.word
```

```
  val shiftr-signed : Word8.word  $\rightarrow$  IntInf.int  $\rightarrow$  Word8.word
```

```
  val test-bit : Word8.word  $\rightarrow$  IntInf.int  $\rightarrow$  bool
```

```
end = struct
```

```
fun set-bit x n b =
```

```
  let val mask = Word8.<< (0wx1, Word.fromLargeInt (IntInf.toLarge n))
```

```
  in if b then Word8.orb (x, mask)
```

```
    else Word8.andb (x, Word8.notb mask)
```

```
  end
```

```
fun shiftl x n =
```

```
  Word8.<< (x, Word.fromLargeInt (IntInf.toLarge n))
```

```
fun shiftr x n =
```

```
  Word8.>> (x, Word.fromLargeInt (IntInf.toLarge n))
```

```
fun shiftr-signed x n =
```

```
  Word8.~>> (x, Word.fromLargeInt (IntInf.toLarge n))
```

```
fun test-bit x n =
```

```
  Word8.andb (x, Word8.<< (0wx1, Word.fromLargeInt (IntInf.toLarge n))) <>  
  Word8.fromInt 0
```

```
end; (* struct Uint8 *)
```

```
code-reserved SML Uint8
```

```
code-printing code-module Uint8  $\rightarrow$  (Haskell)
```

```
<module Uint8(Int8, Word8) where
```

```
  import Data.Int(Int8)
```

```
  import Data.Word(Word8)>
```

```
code-reserved Haskell Uint8
```

Scala provides only signed 8bit numbers, so we use these and implement sign-sensitive operations like comparisons manually.

code-printing code-module *Uint8* \rightarrow (*Scala*)

```

⟨object Uint8 {

def less(x: Byte, y: Byte) : Boolean =
  x < 0 match {
    case true => y < 0 && x < y
    case false => y < 0 || x < y
  }

def less-eq(x: Byte, y: Byte) : Boolean =
  x < 0 match {
    case true => y < 0 && x <= y
    case false => y < 0 || x <= y
  }

def set-bit(x: Byte, n: BigInt, b: Boolean) : Byte =
  b match {
    case true => (x | (1 << n.intValue)).toByte
    case false => (x & (1 << n.intValue).unary~).toByte
  }

def shiftl(x: Byte, n: BigInt) : Byte = (x << n.intValue).toByte

def shiftr(x: Byte, n: BigInt) : Byte = ((x & 255) >>> n.intValue).toByte

def shiftr-signed(x: Byte, n: BigInt) : Byte = (x >> n.intValue).toByte

def test-bit(x: Byte, n: BigInt) : Boolean =
  (x & (1 << n.intValue)) != 0

} /* object Uint8 */
code-reserved Scala Uint8

```

Avoid *Abs-uint8* in generated code, use *Rep-uint8'* instead. The symbolic implementations for `code_simp` use *Rep-uint8*.

The new destructor *Rep-uint8'* is executable. As the simplifier is given the [code abstract] equations literally, we cannot implement *Rep-uint8* directly, because that makes `code_simp` loop.

If code generation raises `Match`, some equation probably contains *Rep-uint8* ([code abstract] equations for *uint8* may use *Rep-uint8* because these instances will be folded away.)

To convert *8 word* values into *uint8*, use *Abs-uint8'*.

definition *Rep-uint8'* **where** [*simp*]: *Rep-uint8'* = *Rep-uint8*

lemma *Rep-uint8'-transfer* [*transfer-rule*]:

```

rel-fun cr-uint8 (=) (λx. x) Rep-uint8'
⟨proof⟩

```

lemma *Rep-uint8'-code* [code]: *Rep-uint8' x = (BITS n. bit x n)*
 ⟨proof⟩

lift-definition *Abs-uint8' :: 8 word ⇒ uint8 is λx :: 8 word. x* ⟨proof⟩

lemma *Abs-uint8'-code* [code]: *Abs-uint8' x = Uint8 (integer-of-int (uint x))*
including *integer.lifting* ⟨proof⟩

declare [[code drop: *term-of-class.term-of :: uint8 ⇒ -*]]

lemma *term-of-uint8-code* [code]:

defines *TR ≡ typerep.TypeRep and bit0 ≡ STR "Numeral-Type.bit0"* **shows**
term-of-class.term-of x =
Code-Evaluation.App (Code-Evaluation.Const (STR "Uint8.uint8.Abs-uint8'")
(TR (STR "fun") [TR (STR "Word.word") [TR bit0 [TR bit0 [TR bit0 [TR (STR
"Numeral-Type.num1'" []]]], TR (STR "Uint8.uint8'") []]))
(term-of-class.term-of (Rep-uint8' x))
 ⟨proof⟩

lemma *Uint8-code* [code]: *Rep-uint8 (Uint8 i) = word-of-int (int-of-integer-symbolic i)*
 ⟨proof⟩

code-printing type-constructor *uint8* →

(SML) *Word8.word* **and**

(Haskell) *Uint8.Word8* **and**

(Scala) *Byte*

| **constant** *Uint8* →

(SML) *Word8.fromLargeInt (IntInf.toLarge -)* **and**

(Haskell) (*Prelude.fromInteger - :: Uint8.Word8*) **and**

(Haskell-Quickcheck) (*Prelude.fromInteger (Prelude.toInteger -) :: Uint8.Word8*)

and

(Scala) *-.byteValue*

| **constant** *0 :: uint8* →

(SML) (*Word8.fromInt 0*) **and**

(Haskell) (*0 :: Uint8.Word8*) **and**

(Scala) *0.toByte*

| **constant** *1 :: uint8* →

(SML) (*Word8.fromInt 1*) **and**

(Haskell) (*1 :: Uint8.Word8*) **and**

(Scala) *1.toByte*

| **constant** *plus :: uint8 ⇒ - ⇒ -* →

(SML) *Word8.+ ((-, (-))* **and**

(Haskell) **infixl 6 +** **and**

(Scala) (*- +/ -*).*toByte*

| **constant** *uminus :: uint8 ⇒ -* →

(SML) *Word8.~* **and**

(Haskell) *negate* **and**

(Scala) (*- -*).*toByte*


```

| constant minus :: uint8 ⇒ - →
  (SML) Word8.- ((-), (-)) and
  (Haskell) infixl 6 - and
  (Scala) (- -/ -).toByte
| constant times :: uint8 ⇒ - ⇒ - →
  (SML) Word8.* ((-), (-)) and
  (Haskell) infixl 7 * and
  (Scala) (- */ -).toByte
| constant HOL.equal :: uint8 ⇒ - ⇒ bool →
  (SML) !((- : Word8.word) = -) and
  (Haskell) infix 4 == and
  (Scala) infixl 5 ==
| class-instance uint8 :: equal → (Haskell) -
| constant less-eq :: uint8 ⇒ - ⇒ bool →
  (SML) Word8.<= ((-), (-)) and
  (Haskell) infix 4 <= and
  (Scala) Uint8.less'-eq
| constant less :: uint8 ⇒ - ⇒ bool →
  (SML) Word8.< ((-), (-)) and
  (Haskell) infix 4 < and
  (Scala) Uint8.less
| constant Bit-Operations.not :: uint8 ⇒ - →
  (SML) Word8.notb and
  (Haskell) Data'-Bits.complement and
  (Scala) -.unary'~.toByte
| constant Bit-Operations.and :: uint8 ⇒ - →
  (SML) Word8.andb ((-),/ (-)) and
  (Haskell) infixl 7 Data-Bits.&. and
  (Scala) (- & -).toByte
| constant Bit-Operations.or :: uint8 ⇒ - →
  (SML) Word8.orb ((-),/ (-)) and
  (Haskell) infixl 5 Data-Bits.|. and
  (Scala) (- | -).toByte
| constant Bit-Operations.xor :: uint8 ⇒ - →
  (SML) Word8.xorb ((-),/ (-)) and
  (Haskell) Data'-Bits.xor and
  (Scala) (- ^ -).toByte

```

definition *uint8-divmod* :: *uint8* ⇒ *uint8* ⇒ *uint8* × *uint8* **where**

```

  uint8-divmod x y =
    (if y = 0 then (undefined ((div) :: uint8 ⇒ -) x (0 :: uint8), undefined ((mod) ::
uint8 ⇒ -) x (0 :: uint8))
    else (x div y, x mod y)

```

definition *uint8-div* :: *uint8* ⇒ *uint8* ⇒ *uint8*
where *uint8-div* *x y* = *fst* (*uint8-divmod* *x y*)

definition *uint8-mod* :: *uint8* ⇒ *uint8* ⇒ *uint8*
where *uint8-mod* *x y* = *snd* (*uint8-divmod* *x y*)

lemma *div-uint8-code* [code]: $x \text{ div } y = (\text{if } y = 0 \text{ then } 0 \text{ else } \text{uint8-div } x \ y)$
including *undefined-transfer* ⟨proof⟩

lemma *mod-uint8-code* [code]: $x \text{ mod } y = (\text{if } y = 0 \text{ then } x \text{ else } \text{uint8-mod } x \ y)$
including *undefined-transfer* ⟨proof⟩

definition *uint8-sdiv* :: $\text{uint8} \Rightarrow \text{uint8} \Rightarrow \text{uint8}$
where

uint8-sdiv $x \ y =$
 (*if* $y = 0$ *then* *undefined* $((\text{div}) :: \text{uint8} \Rightarrow -) \ x \ (0 :: \text{uint8})$
else *Abs-uint8* $(\text{Rep-uint8 } x \ \text{sdiv } \text{Rep-uint8 } y)$)

definition *div0-uint8* :: $\text{uint8} \Rightarrow \text{uint8}$

where [code del]: *div0-uint8* $x = \text{undefined } ((\text{div}) :: \text{uint8} \Rightarrow -) \ x \ (0 :: \text{uint8})$

declare [[code abort: *div0-uint8*]]

definition *mod0-uint8* :: $\text{uint8} \Rightarrow \text{uint8}$

where [code del]: *mod0-uint8* $x = \text{undefined } ((\text{mod}) :: \text{uint8} \Rightarrow -) \ x \ (0 :: \text{uint8})$

declare [[code abort: *mod0-uint8*]]

lemma *uint8-divmod-code* [code]:

uint8-divmod $x \ y =$
 (*if* $0x80 \leq y$ *then* *if* $x < y$ *then* $(0, x)$ *else* $(1, x - y)$
else *if* $y = 0$ *then* $(\text{div0-uint8 } x, \text{mod0-uint8 } x)$
else *let* $q = \text{push-bit } 1 \ (\text{uint8-sdiv } (\text{drop-bit } 1 \ x) \ y);$
 $r = x - q * y$
in *if* $r \geq y$ *then* $(q + 1, r - y)$ *else* (q, r))

including *undefined-transfer* ⟨proof⟩

lemma *uint8-sdiv-code* [code]:

Rep-uint8 $(\text{uint8-sdiv } x \ y) =$
 (*if* $y = 0$ *then* *Rep-uint8* $(\text{undefined } ((\text{div}) :: \text{uint8} \Rightarrow -) \ x \ (0 :: \text{uint8}))$
else *Rep-uint8* $x \ \text{sdiv } \text{Rep-uint8 } y$)

⟨proof⟩

Note that we only need a translation for signed division, but not for the remainder because $\text{uint8-divmod } ?x \ ?y = (\text{if } 128 \leq ?y \text{ then if } ?x < ?y \text{ then } (0, ?x) \text{ else } (1, ?x - ?y) \text{ else if } ?y = 0 \text{ then } (\text{div0-uint8 } ?x, \text{mod0-uint8 } ?x) \text{ else let } q = \text{push-bit } 1 \ (\text{uint8-sdiv } (\text{drop-bit } 1 \ ?x) \ ?y); r = ?x - q * ?y \text{ in if } ?y \leq r \text{ then } (q + 1, r - ?y) \text{ else } (q, r))$ computes both with division only.

code-printing

constant *uint8-div* \rightarrow
 (SML) *Word8.div* $((-), (-))$ **and**
 (Haskell) *Prelude.div*

| **constant** *uint8-mod* \rightarrow
 (SML) *Word8.mod* $((-), (-))$ **and**
 (Haskell) *Prelude.mod*

```
| constant uint8-divmod  $\rightarrow$ 
  (Haskell) divmod
| constant uint8-sdiv  $\rightarrow$ 
  (Scala) (- '/' -).toByte
```

definition *uint8-test-bit* :: *uint8* \Rightarrow *integer* \Rightarrow *bool*
where [*code del*]:
uint8-test-bit *x n* =
 (*if* $n < 0 \vee 7 < n$ *then* *undefined* (*bit* :: *uint8* \Rightarrow -) *x n*
else *bit* *x* (*nat-of-integer* *n*))

lemma *bit-uint8-code* [*code*]:
bit *x n* \longleftrightarrow $n < 8 \wedge$ *uint8-test-bit* *x* (*integer-of-nat* *n*)
including *undefined-transfer integer.lifting* \langle *proof* \rangle

lemma *uint8-test-bit-code* [*code*]:
uint8-test-bit *w n* =
 (*if* $n < 0 \vee 7 < n$ *then* *undefined* (*bit* :: *uint8* \Rightarrow -) *w n* *else* *bit* (*Rep-uint8* *w*)
 (*nat-of-integer* *n*))
 \langle *proof* \rangle

code-printing constant *uint8-test-bit* \rightarrow
 (*SML*) *Uint8.test'-bit* **and**
 (*Haskell*) *Data'-Bits.testBitBounded* **and**
 (*Scala*) *Uint8.test'-bit* **and**
 (*Eval*) (*fn* *x* \Rightarrow *fn* *i* \Rightarrow *if* $i < 0$ *orelse* $i \geq 8$ *then* *raise* (*Fail* *argument to*
uint8'-test'-bit out of bounds) *else* *Uint8.test'-bit* *x i*)

definition *uint8-set-bit* :: *uint8* \Rightarrow *integer* \Rightarrow *bool* \Rightarrow *uint8*
where [*code del*]:
uint8-set-bit *x n b* =
 (*if* $n < 0 \vee 7 < n$ *then* *undefined* (*set-bit* :: *uint8* \Rightarrow -) *x n b*
else *set-bit* *x* (*nat-of-integer* *n*) *b*)

lemma *set-bit-uint8-code* [*code*]:
set-bit *x n b* = (*if* $n < 8$ *then* *uint8-set-bit* *x* (*integer-of-nat* *n*) *b* *else* *x*)
including *undefined-transfer integer.lifting* \langle *proof* \rangle

lemma *uint8-set-bit-code* [*code*]:
Rep-uint8 (*uint8-set-bit* *w n b*) =
 (*if* $n < 0 \vee 7 < n$ *then* *Rep-uint8* (*undefined* (*set-bit* :: *uint8* \Rightarrow -) *w n b*)
else *set-bit* (*Rep-uint8* *w*) (*nat-of-integer* *n*) *b*)
including *undefined-transfer* \langle *proof* \rangle

code-printing constant *uint8-set-bit* \rightarrow
 (*SML*) *Uint8.set'-bit* **and**
 (*Haskell*) *Data'-Bits.setBitBounded* **and**
 (*Scala*) *Uint8.set'-bit* **and**

(Eval) (fn x => fn i => fn b => if i < 0 orelse i >= 8 then raise (Fail argument to uint8'-set'-bit out of bounds) else Uint8.set'-bit x i b)

definition *uint8-shiffl* :: *uint8* ⇒ *integer* ⇒ *uint8*

where [code del]:

uint8-shiffl x n = (if n < 0 ∨ 8 ≤ n then undefined (push-bit :: nat ⇒ uint8 ⇒ -) x n else push-bit (nat-of-integer n) x)

lemma *shiffl-uint8-code* [code]:

push-bit n x = (if n < 8 then *uint8-shiffl* x (integer-of-nat n) else 0)

including undefined-transfer integer.lifting ⟨proof⟩

lemma *uint8-shiffl-code* [code]:

Rep-uint8 (*uint8-shiffl* w n) =

(if n < 0 ∨ 8 ≤ n then Rep-uint8 (undefined (push-bit :: nat ⇒ uint8 ⇒ -) w n) else push-bit (nat-of-integer n) (Rep-uint8 w))

including undefined-transfer ⟨proof⟩

code-printing constant *uint8-shiffl* ↪

(SML) *Uint8.shiffl* **and**

(Haskell) *Data'-Bits.shifflBounded* **and**

(Scala) *Uint8.shiffl* **and**

(Eval) (fn x => fn i => if i < 0 orelse i >= 8 then raise (Fail argument to uint8'-shiffl out of bounds) else *Uint8.shiffl* x i)

definition *uint8-shiftr* :: *uint8* ⇒ *integer* ⇒ *uint8*

where [code del]:

uint8-shiftr x n = (if n < 0 ∨ 8 ≤ n then undefined (drop-bit :: - ⇒ - ⇒ uint8) x n else drop-bit (nat-of-integer n) x)

lemma *shiftr-uint8-code* [code]:

drop-bit n x = (if n < 8 then *uint8-shiftr* x (integer-of-nat n) else 0)

including undefined-transfer integer.lifting ⟨proof⟩

lemma *uint8-shiftr-code* [code]:

Rep-uint8 (*uint8-shiftr* w n) =

(if n < 0 ∨ 8 ≤ n then Rep-uint8 (undefined (drop-bit :: - ⇒ - ⇒ uint8) w n) else drop-bit (nat-of-integer n) (Rep-uint8 w))

including undefined-transfer ⟨proof⟩

code-printing constant *uint8-shiftr* ↪

(SML) *Uint8.shiftr* **and**

(Haskell) *Data'-Bits.shiftrBounded* **and**

(Scala) *Uint8.shiftr* **and**

(Eval) (fn x => fn i => if i < 0 orelse i >= 8 then raise (Fail argument to uint8'-shiftr out of bounds) else *Uint8.shiftr* x i)

definition *uint8-sshiftr* :: *uint8* ⇒ *integer* ⇒ *uint8*

where `[code del]`:

```
uint8-sshiftr x n =
  (if n < 0 ∨ 8 ≤ n then undefined signed-drop-bit-uint8 n x else signed-drop-bit-uint8
  (nat-of-integer n) x)
```

lemma `sshiftr-uint8-code` `[code]`:

```
signed-drop-bit-uint8 n x =
  (if n < 8 then uint8-sshiftr x (integer-of-nat n) else if bit x 7 then -1 else 0)
including undefined-transfer integer.lifting <proof>
```

lemma `uint8-sshiftr-code` `[code]`:

```
Rep-uint8 (uint8-sshiftr w n) =
  (if n < 0 ∨ 8 ≤ n then Rep-uint8 (undefined signed-drop-bit-uint8 n w)
  else signed-drop-bit (nat-of-integer n) (Rep-uint8 w))
including undefined-transfer <proof>
```

code-printing constant `uint8-sshiftr` \rightarrow

```
(SML) Uint8.shiftr'-signed and
(Haskell)
  (Prelude.fromInteger (Prelude.toInteger (Data'-Bits.shiftrBounded (Prelude.fromInteger
  (Prelude.toInteger -) :: Uint8.Int8) -)) :: Uint8.Word8) and
(Scala) Uint8.shiftr'-signed and
(Eval) (fn x => fn i => if i < 0 orelse i ≥ 8 then raise (Fail argument to
uint8'-sshiftr out of bounds) else Uint8.shiftr'-signed x i)
```

context

includes `bit-operations-syntax`

begin

lemma `uint8-msb-test-bit`: $msb\ x \longleftrightarrow bit\ (x :: uint8)\ 7$
`<proof>`

lemma `msb-uint16-code` `[code]`: $msb\ x \longleftrightarrow uint8-test-bit\ x\ 7$
`<proof>`

lemma `uint8-of-int-code` `[code]`:

```
uint8-of-int i = Uint8 (integer-of-int i)
including integer.lifting <proof>
```

lemma `int-of-uint8-code` `[code]`:

```
int-of-uint8 x = int-of-integer (integer-of-uint8 x)
<proof>
```

lemma `uint8-of-nat-code` `[code]`:

```
uint8-of-nat = uint8-of-int ∘ int
<proof>
```

lemma `nat-of-uint8-code` `[code]`:

```
nat-of-uint8 x = nat-of-integer (integer-of-uint8 x)
```

<proof> **including** *integer.lifting* *<proof>*

definition *integer-of-uint8-signed* :: *uint8* ⇒ *integer*

where

integer-of-uint8-signed *n* = (if bit *n* 7 then undefined *integer-of-uint8* *n* else *integer-of-uint8* *n*)

lemma *integer-of-uint8-signed-code* [*code*]:

integer-of-uint8-signed *n* =
 (if bit *n* 7 then undefined *integer-of-uint8* *n* else *integer-of-int* (*uint* (*Rep-uint8*'
n)))
<proof>

lemma *integer-of-uint8-code* [*code*]:

integer-of-uint8 *n* =
 (if bit *n* 7 then *integer-of-uint8-signed* (*n* AND 0x7F) OR 0x80 else *integer-of-uint8-signed*
n)
<proof>
including *integer.lifting* *<proof>*

end

code-printing

constant *integer-of-uint8* →
 (*SML*) *IntInf.fromLarge* (*Word8.toLargeInt* -) **and**
 (*Haskell*) *Prelude.toInteger*
 | **constant** *integer-of-uint8-signed* →
 (*Scala*) *BigInt*

8.3 Quickcheck setup

definition *uint8-of-natural* :: *natural* ⇒ *uint8*

where *uint8-of-natural* *x* ≡ *UInt8* (*integer-of-natural* *x*)

instantiation *uint8* :: {*random*, *exhaustive*, *full-exhaustive*} **begin**

definition *random-uint8* ≡ *qc-random-cnv* *uint8-of-natural*

definition *exhaustive-uint8* ≡ *qc-exhaustive-cnv* *uint8-of-natural*

definition *full-exhaustive-uint8* ≡ *qc-full-exhaustive-cnv* *uint8-of-natural*

instance *<proof>*

end

instantiation *uint8* :: *narrowing* **begin**

interpretation *quickcheck-narrowing-samples*

λ*i*. let *x* = *UInt8* *i* in (*x*, 0xFF - *x*) 0
Typerep.Typerep (*STR* "UInt8.uint8") [] *<proof>*

definition *narrowing-uint8* *d* = *qc-narrowing-drawn-from* (*narrowing-samples* *d*)
d

```
declare [[code drop: partial-term-of :: uint8 itself ⇒ -]]  
lemmas partial-term-of-uint8 [code] = partial-term-of-code  
  
instance ⟨proof⟩  
end  
  
end
```


Chapter 9

Unsigned words of default size

```
theory Uint imports  
  Word-Type-Copies  
  Code-Target-Integer-Bit  
begin
```

This theory provides access to words in the target languages of the code generator whose bit width is the default of the target language. To that end, the type *uint* models words of width *dflt-size*, but *dflt-size* is known only to be positive.

Usage restrictions: Default-size words (type *uint*) cannot be used for evaluation, because the results depend on the particular choice of word size in the target language and implementation. Symbolic evaluation has not yet been set up for *uint*.

The default size type

```
typedecl dflt-size
```

```
instantiation dflt-size :: typerep begin
```

```
definition typerep-class.typerep  $\equiv$   $\lambda-$  :: dflt-size itself. Typerep.TypeRep (STR  
"Uint.dflt-size'") []
```

```
instance  $\langle$ proof $\rangle$ 
```

```
end
```

```
consts dflt-size-aux :: nat
```

```
specification (dflt-size-aux) dflt-size-aux-g0: dflt-size-aux > 0  
 $\langle$ proof $\rangle$ 
```

```
hide-fact dflt-size-aux-def
```

```
instantiation dflt-size :: len begin
```

```
definition len-of-dflt-size ( $-$  :: dflt-size itself)  $\equiv$  dflt-size-aux
```

```
instance  $\langle$ proof $\rangle$ 
```

end

abbreviation *dflt-size* \equiv *len-of* (*TYPE* (*dflt-size*))

context includes *integer.lifting* **begin**

lift-definition *dflt-size-integer* :: *integer* **is** *int dflt-size* \langle *proof* \rangle

declare *dflt-size-integer-def* [*code del*]

— The code generator will substitute a machine-dependent value for this constant

lemma *dflt-size-by-int* [*code*]: *dflt-size* = *nat-of-integer dflt-size-integer*
 \langle *proof* \rangle

lemma *dflt-size* [*simp*]:

dflt-size > 0

dflt-size \geq *Suc* 0

\neg *dflt-size* < *Suc* 0

\langle *proof* \rangle

end

9.1 Type definition and primitive operations

typedef *uint* = \langle *UNIV* :: *dflt-size word set* \rangle \langle *proof* \rangle

global-interpretation *uint*: *word-type-copy Abs-uint Rep-uint*
 \langle *proof* \rangle

setup-lifting *type-definition-uint*

declare *uint.of-word-of* [*code abstype*]

declare *Quotient-uint* [*transfer-rule*]

instantiation *uint* :: \langle {*comm-ring-1*, *semiring-modulo*, *equal*, *linorder*} \rangle
begin

lift-definition *zero-uint* :: *uint* **is** 0 \langle *proof* \rangle

lift-definition *one-uint* :: *uint* **is** 1 \langle *proof* \rangle

lift-definition *plus-uint* :: \langle *uint* \Rightarrow *uint* \Rightarrow *uint* \rangle **is** \langle (+) \rangle \langle *proof* \rangle

lift-definition *uminus-uint* :: \langle *uint* \Rightarrow *uint* \rangle **is** *uminus* \langle *proof* \rangle

lift-definition *minus-uint* :: \langle *uint* \Rightarrow *uint* \Rightarrow *uint* \rangle **is** \langle (-) \rangle \langle *proof* \rangle

lift-definition *times-uint* :: \langle *uint* \Rightarrow *uint* \Rightarrow *uint* \rangle **is** \langle (*) \rangle \langle *proof* \rangle

lift-definition *divide-uint* :: \langle *uint* \Rightarrow *uint* \Rightarrow *uint* \rangle **is** \langle (div) \rangle \langle *proof* \rangle

lift-definition *modulo-uint* :: \langle *uint* \Rightarrow *uint* \Rightarrow *uint* \rangle **is** \langle (mod) \rangle \langle *proof* \rangle

lift-definition *equal-uint* :: \langle *uint* \Rightarrow *uint* \Rightarrow *bool* \rangle **is** \langle *HOL.equal* \rangle \langle *proof* \rangle

lift-definition *less-eq-uint* :: \langle *uint* \Rightarrow *uint* \Rightarrow *bool* \rangle **is** \langle (\leq) \rangle \langle *proof* \rangle

lift-definition *less-uint* :: \langle *uint* \Rightarrow *uint* \Rightarrow *bool* \rangle **is** \langle (<) \rangle \langle *proof* \rangle

global-interpretation *uint*: *word-type-copy-ring Abs-uint Rep-uint*
 \langle *proof* \rangle

instance $\langle proof \rangle$

end

instantiation *uint* :: *ring-bit-operations*
begin

lift-definition *bit-uint* :: $\langle uint \Rightarrow nat \Rightarrow bool \rangle$ **is** *bit* $\langle proof \rangle$
lift-definition *not-uint* :: $\langle uint \Rightarrow uint \rangle$ **is** $\langle Bit-Operations.not \rangle$ $\langle proof \rangle$
lift-definition *and-uint* :: $\langle uint \Rightarrow uint \Rightarrow uint \rangle$ **is** $\langle Bit-Operations.and \rangle$ $\langle proof \rangle$
lift-definition *or-uint* :: $\langle uint \Rightarrow uint \Rightarrow uint \rangle$ **is** $\langle Bit-Operations.or \rangle$ $\langle proof \rangle$
lift-definition *xor-uint* :: $\langle uint \Rightarrow uint \Rightarrow uint \rangle$ **is** $\langle Bit-Operations.xor \rangle$ $\langle proof \rangle$
lift-definition *mask-uint* :: $\langle nat \Rightarrow uint \rangle$ **is** *mask* $\langle proof \rangle$
lift-definition *push-bit-uint* :: $\langle nat \Rightarrow uint \Rightarrow uint \rangle$ **is** *push-bit* $\langle proof \rangle$
lift-definition *drop-bit-uint* :: $\langle nat \Rightarrow uint \Rightarrow uint \rangle$ **is** *drop-bit* $\langle proof \rangle$
lift-definition *signed-drop-bit-uint* :: $\langle nat \Rightarrow uint \Rightarrow uint \rangle$ **is** *signed-drop-bit* $\langle proof \rangle$
lift-definition *take-bit-uint* :: $\langle nat \Rightarrow uint \Rightarrow uint \rangle$ **is** *take-bit* $\langle proof \rangle$
lift-definition *set-bit-uint* :: $\langle nat \Rightarrow uint \Rightarrow uint \rangle$ **is** $\langle Bit-Operations.set-bit \rangle$ $\langle proof \rangle$
lift-definition *unset-bit-uint* :: $\langle nat \Rightarrow uint \Rightarrow uint \rangle$ **is** *unset-bit* $\langle proof \rangle$
lift-definition *flip-bit-uint* :: $\langle nat \Rightarrow uint \Rightarrow uint \rangle$ **is** *flip-bit* $\langle proof \rangle$

global-interpretation *uint*: *word-type-copy-bits Abs-uint Rep-uint signed-drop-bit-uint*
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

lift-definition *uint-of-nat* :: $\langle nat \Rightarrow uint \rangle$
is *word-of-nat* $\langle proof \rangle$

lift-definition *nat-of-uint* :: $\langle uint \Rightarrow nat \rangle$
is *unat* $\langle proof \rangle$

lift-definition *uint-of-int* :: $\langle int \Rightarrow uint \rangle$
is *word-of-int* $\langle proof \rangle$

lift-definition *int-of-uint* :: $\langle uint \Rightarrow int \rangle$
is *uint* $\langle proof \rangle$

context
includes *integer.lifting*
begin

lift-definition *Uint* :: $\langle integer \Rightarrow uint \rangle$
is *word-of-int* $\langle proof \rangle$

lift-definition *integer-of-uint* :: $\langle \text{uint} \Rightarrow \text{integer} \rangle$
is *uint* $\langle \text{proof} \rangle$

end

global-interpretation *uint*: *word-type-copy-more Abs-uint Rep-uint signed-drop-bit-uint*
uint-of-nat nat-of-uint uint-of-int int-of-uint Uint integer-of-uint
 $\langle \text{proof} \rangle$

instantiation *uint* :: $\{ \text{size}, \text{msb}, \text{lsb}, \text{set-bit}, \text{bit-comprehension} \}$
begin

lift-definition *size-uint* :: $\langle \text{uint} \Rightarrow \text{nat} \rangle$ **is** *size* $\langle \text{proof} \rangle$

lift-definition *msb-uint* :: $\langle \text{uint} \Rightarrow \text{bool} \rangle$ **is** *msb* $\langle \text{proof} \rangle$

lift-definition *lsb-uint* :: $\langle \text{uint} \Rightarrow \text{bool} \rangle$ **is** *lsb* $\langle \text{proof} \rangle$

Workaround: avoid name space clash by spelling out *lift-definition* explicitly.

definition *set-bit-uint* :: $\langle \text{uint} \Rightarrow \text{nat} \Rightarrow \text{bool} \Rightarrow \text{uint} \rangle$
where *set-bit-uint-eq*: $\langle \text{set-bit-uint } a \ n \ b = (\text{if } b \ \text{then } \text{Bit-Operations.set-bit} \ \text{else } \text{unset-bit}) \ n \ a \rangle$

context

includes *lifting-syntax*

begin

lemma *set-bit-uint-transfer* [*transfer-rule*]:

$\langle (cr\text{-uint} \implies) (=) \implies (\longleftrightarrow) \implies cr\text{-uint} \rangle \text{Generic-set-bit.set-bit Generic-set-bit.set-bit}$
 $\langle \text{proof} \rangle$

end

lift-definition *set-bits-uint* :: $\langle (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{uint} \rangle$ **is** *set-bits* $\langle \text{proof} \rangle$

lift-definition *set-bits-aux-uint* :: $\langle (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow \text{uint} \Rightarrow \text{uint} \rangle$ **is** *set-bits-aux*
 $\langle \text{proof} \rangle$

global-interpretation *uint*: *word-type-copy-misc Abs-uint Rep-uint signed-drop-bit-uint*
uint-of-nat nat-of-uint uint-of-int int-of-uint Uint integer-of-uint dflt-size set-bits-aux-uint
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

9.2 Code setup

code-printing code-module *Uint* \rightarrow (*SML*)

\langle

structure Uint : *sig*

```

    val set-bit : Word.word -> IntInf.int -> bool -> Word.word
    val shiftl : Word.word -> IntInf.int -> Word.word
    val shiftr : Word.word -> IntInf.int -> Word.word
    val shiftr-signed : Word.word -> IntInf.int -> Word.word
    val test-bit : Word.word -> IntInf.int -> bool
end = struct

fun set-bit x n b =
  let val mask = Word.<< (0wx1, Word.fromLargeInt (IntInf.toLarge n))
  in if b then Word.orb (x, mask)
    else Word.andb (x, Word.notb mask)
  end

fun shiftl x n =
  Word.<< (x, Word.fromLargeInt (IntInf.toLarge n))

fun shiftr x n =
  Word.>> (x, Word.fromLargeInt (IntInf.toLarge n))

fun shiftr-signed x n =
  Word.~>> (x, Word.fromLargeInt (IntInf.toLarge n))

fun test-bit x n =
  Word.andb (x, Word.<< (0wx1, Word.fromLargeInt (IntInf.toLarge n))) <>
  Word.fromInt 0

end; (* struct Uint *)
code-reserved SML Uint

code-printing code-module Uint  $\rightarrow$  (Haskell)
<module Uint(Int, Word, dflt-size) where

  import qualified Prelude
  import Data.Int(Int)
  import Data.Word(Word)
  import qualified Data.Bits

  dflt-size :: Prelude.Integer
  dflt-size = Prelude.toInteger (bitSize-aux (0::Word)) where
    bitSize-aux :: (Data.Bits.Bits a, Prelude.Bounded a) => a -> Int
    bitSize-aux = Data.Bits.bitSize
  and (Haskell-Quickcheck)
  <module Uint(Int, Word, dflt-size) where

    import qualified Prelude
    import Data.Int(Int)
    import Data.Word(Word)
    import qualified Data.Bits

```

```

dflt-size :: Prelude.Int
dflt-size = bitSize-aux (0::Word) where
  bitSize-aux :: (Data.Bits.Bits a, Prelude.Bounded a) => a -> Int
  bitSize-aux = Data.Bits.bitSize
>
code-reserved Haskell Uint dflt-size

```

OCaml and Scala provide only signed bit numbers, so we use these and implement sign-sensitive operations like comparisons manually.

code-printing code-module *Uint* \rightarrow (OCaml)

```

<module Uint : sig
  type t = int
  val dflt-size : Z.t
  val less : t -> t -> bool
  val less-eq : t -> t -> bool
  val set-bit : t -> Z.t -> bool -> t
  val shiftl : t -> Z.t -> t
  val shiftr : t -> Z.t -> t
  val shiftr-signed : t -> Z.t -> t
  val test-bit : t -> Z.t -> bool
  val int-mask : int
  val int32-mask : int32
  val int64-mask : int64
end = struct

type t = int

let dflt-size = Z.of-int Sys.int-size;;

(* negative numbers have their highest bit set,
   so they are greater than positive ones *)
let less x y =
  if x < 0 then
    y < 0 && x < y
  else y < 0 || x < y;;

let less-eq x y =
  if x < 0 then
    y < 0 && x <= y
  else y < 0 || x <= y;;

let set-bit x n b =
  let mask = 1 lsl (Z.to-int n)
  in if b then x lor mask
     else x land (lnot mask);;

let shiftl x n = x lsl (Z.to-int n);;

let shiftr x n = x lsr (Z.to-int n);;

```

```

let shiftr-signed x n = x asr (Z.to-int n);

let test-bit x n = x land (1 lsl (Z.to-int n)) <> 0;;

let int-mask =
  if Sys.int-size < 32 then lnot 0 else 0xFFFFFFFF;;

let int32-mask =
  if Sys.int-size < 32 then Int32.pred (Int32.shift-left Int32.one Sys.int-size)
  else Int32.of-string 0xFFFFFFFF;;

let int64-mask =
  if Sys.int-size < 64 then Int64.pred (Int64.shift-left Int64.one Sys.int-size)
  else Int64.of-string 0xFFFFFFFFFFFFFFFF;;

end;; (*struct Uint*)
code-reserved OCaml Uint

code-printing code-module Uint → (Scala)
⟨object Uint {
  def dflt-size : BigInt = BigInt(32)

  def less(x: Int, y: Int) : Boolean =
    x < 0 match {
      case true => y < 0 && x < y
      case false => y < 0 || x < y
    }

  def less-eq(x: Int, y: Int) : Boolean =
    x < 0 match {
      case true => y < 0 && x <= y
      case false => y < 0 || x <= y
    }

  def set-bit(x: Int, n: BigInt, b: Boolean) : Int =
    b match {
      case true => x | (1 << n.toIntValue)
      case false => x & (1 << n.toIntValue).unary~
    }

  def shiftl(x: Int, n: BigInt) : Int = x << n.toIntValue

  def shiftr(x: Int, n: BigInt) : Int = x >>> n.toIntValue

  def shiftr-signed(x: Int, n: BigInt) : Int = x >> n.toIntValue

  def test-bit(x: Int, n: BigInt) : Boolean =
    (x & (1 << n.toIntValue)) != 0

```

```

} /* object Uint */
code-reserved Scala Uint

```

OCaml's conversion from `Big_int` to `int` demands that the value fits into a signed integer. The following justifies the implementation.

context

includes *integer.lifting bit-operations-syntax*

begin

definition *wivs-mask* :: *int* **where** *wivs-mask* = $2^{\text{dft-size} - 1}$

lift-definition *wivs-mask-integer* :: *integer* **is** *wivs-mask* *<proof>*

lemma [*code*]: *wivs-mask-integer* = $2^{\text{dft-size} - 1}$
<proof>

definition *wivs-shift* :: *int* **where** *wivs-shift* = $2^{\text{dft-size}}$

lift-definition *wivs-shift-integer* :: *integer* **is** *wivs-shift* *<proof>*

lemma [*code*]: *wivs-shift-integer* = $2^{\text{dft-size}}$
<proof>

definition *wivs-index* :: *nat* **where** *wivs-index* == $\text{dft-size} - 1$

lift-definition *wivs-index-integer* :: *integer* **is** *int wivs-index* *<proof>*

lemma *wivs-index-integer-code*[*code*]: *wivs-index-integer* = $\text{dft-size-integer} - 1$
<proof>

definition *wivs-overflow* :: *int* **where** *wivs-overflow* == $2^{(\text{dft-size} - 1)}$

lift-definition *wivs-overflow-integer* :: *integer* **is** *wivs-overflow* *<proof>*

lemma [*code*]: *wivs-overflow-integer* = $2^{(\text{dft-size} - 1)}$
<proof>

definition *wivs-least* :: *int* **where** *wivs-least* == $- \text{wivs-overflow}$

lift-definition *wivs-least-integer* :: *integer* **is** *wivs-least* *<proof>*

lemma [*code*]: *wivs-least-integer* = $-(2^{(\text{dft-size} - 1)})$
<proof>

definition *Uint-signed* :: *integer* \Rightarrow *uint* **where**

Uint-signed *i* = (if *i* < *wivs-least-integer* \vee *wivs-overflow-integer* \leq *i* then *undefined Uint i* else *Uint i*)

lemma *Uint-code* [*code*]:

Uint *i* =

(let *i'* = *i* AND *wivs-mask-integer* in

if bit *i'* *wivs-index* then *Uint-signed* (*i' - wivs-shift-integer*) else *Uint-signed* *i'*)

including *undefined-transfer*

<proof>

lemma *Uint-signed-code* [*code*]:

Rep-uint (*Uint-signed* *i*) =

(if *i* < *wivs-least-integer* \vee *i* \geq *wivs-overflow-integer* then *Rep-uint* (*undefined*


```

Uint i) else word-of-int (int-of-integer-symbolic i)
  ⟨proof⟩
end

```

Avoid *Abs-uint* in generated code, use *Rep-uint'* instead. The symbolic implementations for `code_simp` use *Rep-uint*.

The new destructor *Rep-uint'* is executable. As the simplifier is given the [code abstract] equations literally, we cannot implement *Rep-uint* directly, because that makes `code_simp` loop.

If code generation raises `Match`, some equation probably contains *Rep-uint* ([code abstract] equations for *uint* may use *Rep-uint* because these instances will be folded away.)

definition *Rep-uint'* **where** [*simp*]: *Rep-uint'* = *Rep-uint*

lemma *Rep-uint'-code* [*code*]: *Rep-uint'* *x* = (*BITS n. bit x n*)
 ⟨*proof*⟩

lift-definition *Abs-uint'* :: *dflt-size word* ⇒ *uint* **is** λ*x* :: *dflt-size word. x* ⟨*proof*⟩

lemma *Abs-uint'-code* [*code*]:
Abs-uint' *x* = *Uint (integer-of-int (uint x))*
including *integer.lifting* ⟨*proof*⟩

declare [[*code drop: term-of-class.term-of* :: *uint* ⇒ -]]

lemma *term-of-uint-code* [*code*]:
defines *TR* ≡ *typerep.TypeRep* **and** *bit0* ≡ *STR "Numeral-Type.bit0"*
shows
term-of-class.term-of x =
Code-Evaluation.App (Code-Evaluation.Const (STR "Uint.uint.Abs-uint')) (TR
(STR "fun") [TR (STR "Word.word") [TR (STR "Uint.dflt-size") []], TR (STR
"Uint.uint") []])
(term-of-class.term-of (Rep-uint' x))
 ⟨*proof*⟩

Important: We must prevent the reflection oracle (`eval-tac`) to use our machine-dependent type.

code-printing

```

type-constructor uint ↪
  (SML) Word.word and
  (Haskell) Uint.Word and
  (OCaml) Uint.t and
  (Scala) Int and
  (Eval) *** Error: Machine dependent type *** and
  (Quickcheck) Word.word
| constant dflt-size-integer ↪
  (SML) (IntInf.fromLarge (Int.toLarge Word.wordSize)) and
  (Eval) (raise (Fail Machine dependent code)) and

```

```

(Quickcheck) Word.wordSize and
(Haskell) Uint.dft'-size and
(OCaml) Uint.dft'-size and
(Scala) Uint.dft'-size
| constant Uint  $\rightarrow$ 
(SML) Word.fromLargeInt (IntInf.toLarge -) and
(Eval) (raise (Fail Machine dependent code)) and
(Quickcheck) Word.fromInt and
(Haskell) (Prelude.fromInteger - :: Uint.Word) and
(Haskell-Quickcheck) (Prelude.fromInteger (Prelude.toInteger -) :: Uint.Word)
and
(Scala) -.intValue
| constant Uint-signed  $\rightarrow$ 
(OCaml) Z.to'-int
| constant 0 :: uint  $\rightarrow$ 
(SML) (Word.fromInt 0) and
(Eval) (raise (Fail Machine dependent code)) and
(Quickcheck) (Word.fromInt 0) and
(Haskell) (0 :: Uint.Word) and
(OCaml) 0 and
(Scala) 0
| constant 1 :: uint  $\rightarrow$ 
(SML) (Word.fromInt 1) and
(Eval) (raise (Fail Machine dependent code)) and
(Quickcheck) (Word.fromInt 1) and
(Haskell) (1 :: Uint.Word) and
(OCaml) 1 and
(Scala) 1
| constant plus :: uint  $\Rightarrow$  -  $\rightarrow$ 
(SML) Word.+ ((-), (-)) and
(Eval) (raise (Fail Machine dependent code)) and
(Quickcheck) Word.+ ((-), (-)) and
(Haskell) infixl 6 + and
(OCaml) Pervasives.(+) and
(Scala) infixl 7 +
| constant uminus :: uint  $\Rightarrow$  -  $\rightarrow$ 
(SML) Word.~ and
(Eval) (raise (Fail Machine dependent code)) and
(Quickcheck) Word.~ and
(Haskell) negate and
(OCaml) Pervasives.(~ -) and
(Scala) !(- -)
| constant minus :: uint  $\Rightarrow$  -  $\rightarrow$ 
(SML) Word.- ((-), (-)) and
(Eval) (raise (Fail Machine dependent code)) and
(Quickcheck) Word.- ((-), (-)) and
(Haskell) infixl 6 - and
(OCaml) Pervasives.(-) and
(Scala) infixl 7 -

```

```

| constant times :: uint ⇒ - ⇒ - →
  (SML) Word.* ((-), (-)) and
  (Eval) (raise (Fail Machine dependent code)) and
  (Quickcheck) Word.* ((-), (-)) and
  (Haskell) infixl 7 * and
  (OCaml) Pervasives.( * ) and
  (Scala) infixl 8 *
| constant HOL.equal :: uint ⇒ - ⇒ bool →
  (SML) !((- : Word.word) = -) and
  (Eval) (raise (Fail Machine dependent code)) and
  (Quickcheck) !((- : Word.word) = -) and
  (Haskell) infix 4 == and
  (OCaml) (Pervasives.(=):Uint.t -> Uint.t -> bool) and
  (Scala) infixl 5 ==
| class-instance uint :: equal →
  (Haskell) -
| constant less-eq :: uint ⇒ - ⇒ bool →
  (SML) Word.<= ((-), (-)) and
  (Eval) (raise (Fail Machine dependent code)) and
  (Quickcheck) Word.<= ((-), (-)) and
  (Haskell) infix 4 <= and
  (OCaml) Uint.less'-eq and
  (Scala) Uint.less'-eq
| constant less :: uint ⇒ - ⇒ bool →
  (SML) Word.< ((-), (-)) and
  (Eval) (raise (Fail Machine dependent code)) and
  (Quickcheck) Word.< ((-), (-)) and
  (Haskell) infix 4 < and
  (OCaml) Uint.less and
  (Scala) Uint.less
| constant Bit-Operations.not :: uint ⇒ - →
  (SML) Word.notb and
  (Eval) (raise (Fail Machine dependent code)) and
  (Quickcheck) Word.notb and
  (Haskell) Data'-Bits.complement and
  (OCaml) Pervasives.lnot and
  (Scala) -.unary'~
| constant Bit-Operations.and :: uint ⇒ - →
  (SML) Word.andb ((-),/ (-)) and
  (Eval) (raise (Fail Machine dependent code)) and
  (Quickcheck) Word.andb ((-),/ (-)) and
  (Haskell) infixl 7 Data-Bits.&. and
  (OCaml) Pervasives.(land) and
  (Scala) infixl 3 &
| constant Bit-Operations.or :: uint ⇒ - →
  (SML) Word.orb ((-),/ (-)) and
  (Eval) (raise (Fail Machine dependent code)) and
  (Quickcheck) Word.orb ((-),/ (-)) and
  (Haskell) infixl 5 Data-Bits.||. and

```

(OCaml) *Pervasives*.(*lor*) **and**
 (Scala) **infixl 1** |
 | **constant** *Bit-Operations*.*xor* :: *uint* \Rightarrow - \rightarrow
 (SML) *Word*.*xorb* ((-),/ (-)) **and**
 (Eval) (*raise* (*Fail Machine dependent code*)) **and**
 (Quickcheck) *Word*.*xorb* ((-),/ (-)) **and**
 (Haskell) *Data'-Bits*.*xor* **and**
 (OCaml) *Pervasives*.(*lxor*) **and**
 (Scala) **infixl 2** ^

definition *uint-divmod* :: *uint* \Rightarrow *uint* \Rightarrow *uint* \times *uint* **where**

uint-divmod *x y* =
 (if *y* = 0 then (undefined ((*div*) :: *uint* \Rightarrow -) *x* (0 :: *uint*), undefined ((*mod*) ::
uint \Rightarrow -) *x* (0 :: *uint*))
 else (*x div y*, *x mod y*)

definition *uint-div* :: *uint* \Rightarrow *uint* \Rightarrow *uint*
where *uint-div* *x y* = *fst* (*uint-divmod* *x y*)

definition *uint-mod* :: *uint* \Rightarrow *uint* \Rightarrow *uint*
where *uint-mod* *x y* = *snd* (*uint-divmod* *x y*)

lemma *div-uint-code* [*code*]: *x div y* = (if *y* = 0 then 0 else *uint-div* *x y*)
including *undefined-transfer* \langle *proof* \rangle

lemma *mod-uint-code* [*code*]: *x mod y* = (if *y* = 0 then *x* else *uint-mod* *x y*)
including *undefined-transfer* \langle *proof* \rangle

definition *uint-sdiv* :: *uint* \Rightarrow *uint* \Rightarrow *uint*
where [*code del*]:
uint-sdiv *x y* =
 (if *y* = 0 then undefined ((*div*) :: *uint* \Rightarrow -) *x* (0 :: *uint*)
 else *Abs-uint* (*Rep-uint* *x sdiv Rep-uint y*)

definition *div0-uint* :: *uint* \Rightarrow *uint*
where [*code del*]: *div0-uint* *x* = undefined ((*div*) :: *uint* \Rightarrow -) *x* (0 :: *uint*)
declare [[*code abort: div0-uint*]]

definition *mod0-uint* :: *uint* \Rightarrow *uint*
where [*code del*]: *mod0-uint* *x* = undefined ((*mod*) :: *uint* \Rightarrow -) *x* (0 :: *uint*)
declare [[*code abort: mod0-uint*]]

definition *wivs-overflow-uint* :: *uint*
where *wivs-overflow-uint* \equiv *push-bit* (*dflt-size* - 1) 1

lemma *Rep-uint-wivs-overflow-uint-eq*:
 \langle *Rep-uint wivs-overflow-uint* = 2^{\wedge} (*dflt-size* - *Suc* 0) \rangle
 \langle *proof* \rangle

lemma *wivs-overflow-uint-greater-eq-0*:

<wivs-overflow-uint > 0>
<proof>

lemma *uint-divmod-code* [code]:

uint-divmod x y =
(if wivs-overflow-uint ≤ y then if x < y then (0, x) else (1, x - y)
else if y = 0 then (div0-uint x, mod0-uint x)
else let q = push-bit 1 (uint-sdiv (drop-bit 1 x) y);
*r = x - q * y*
in if r ≥ y then (q + 1, r - y) else (q, r))
<proof>
including *undefined-transfer*
<proof>

lemma *uint-sdiv-code* [code]:

Rep-uint (uint-sdiv x y) =
(if y = 0 then Rep-uint (undefined ((div) :: uint ⇒ -) x (0 :: uint))
else Rep-uint x sdiv Rep-uint y)
<proof>

Note that we only need a translation for signed division, but not for the remainder because *uint-divmod ?x ?y = (if wivs-overflow-uint ≤ ?y then if ?x < ?y then (0, ?x) else (1, ?x - ?y) else if ?y = 0 then (div0-uint ?x, mod0-uint ?x) else let q = push-bit 1 (uint-sdiv (drop-bit 1 ?x) ?y); r = ?x - q * ?y in if ?y ≤ r then (q + 1, r - ?y) else (q, r))* computes both with division only.

code-printing

constant *uint-div* \rightarrow
(SML) Word.div ((-), (-)) and
(Eval) (raise (Fail Machine dependent code)) and
(Quickcheck) Word.div ((-), (-)) and
(Haskell) Prelude.div
| **constant** *uint-mod* \rightarrow
(SML) Word.mod ((-), (-)) and
(Eval) (raise (Fail Machine dependent code)) and
(Quickcheck) Word.mod ((-), (-)) and
(Haskell) Prelude.mod
| **constant** *uint-divmod* \rightarrow
(Haskell) divmod
| **constant** *uint-sdiv* \rightarrow
(OCaml) Pervasives.(/) and
(Scala) - '/' -

definition *uint-test-bit* :: *uint ⇒ integer ⇒ bool*

where [code del]:

uint-test-bit x n =
(if n < 0 ∨ dflt-size-integer ≤ n then undefined (bit :: uint ⇒ -) x n
else bit x (nat-of-integer n))

lemma *test-bit-uint-code* [code]:
 $bit\ x\ n \iff n < dflt\text{-}size \wedge uint\text{-}test\text{-}bit\ x\ (integer\text{-}of\text{-}nat\ n)$
including *undefined-transfer integer.lifting* ⟨proof⟩

lemma *uint-test-bit-code* [code]:
 $uint\text{-}test\text{-}bit\ w\ n =$
(if $n < 0 \vee dflt\text{-}size\text{-}integer \leq n$ *then* $undefined\ (bit\ ::\ uint \Rightarrow -)$ $w\ n$ *else* bit
 $(Rep\text{-}uint\ w)\ (nat\text{-}of\text{-}integer\ n)$
 ⟨proof⟩

code-printing constant *uint-test-bit* \rightarrow
 (SML) *Uint.test'-bit* **and**
 (Eval) *(raise (Fail Machine dependent code))* **and**
 (Quickcheck) *Uint.test'-bit* **and**
 (Haskell) *Data'-Bits.testBitBounded* **and**
 (OCaml) *Uint.test'-bit* **and**
 (Scala) *Uint.test'-bit*

definition *uint-set-bit* $::\ uint \Rightarrow integer \Rightarrow bool \Rightarrow uint$

where [code del]:

$uint\text{-}set\text{-}bit\ x\ n\ b =$
(if $n < 0 \vee dflt\text{-}size\text{-}integer \leq n$ *then* $undefined\ (set\text{-}bit\ ::\ uint \Rightarrow -)$ $x\ n\ b$
else $set\text{-}bit\ x\ (nat\text{-}of\text{-}integer\ n)\ b$)

lemma *set-bit-uint-code* [code]:
 $set\text{-}bit\ x\ n\ b = (if\ n < dflt\text{-}size\ then\ uint\text{-}set\text{-}bit\ x\ (integer\text{-}of\text{-}nat\ n)\ b\ else\ x)$
including *undefined-transfer integer.lifting* ⟨proof⟩

lemma *uint-set-bit-code* [code]:
 $Rep\text{-}uint\ (uint\text{-}set\text{-}bit\ w\ n\ b) =$
(if $n < 0 \vee dflt\text{-}size\text{-}integer \leq n$ *then* $Rep\text{-}uint\ (undefined\ (set\text{-}bit\ ::\ uint \Rightarrow -)\ w$
 $n\ b)$
else $set\text{-}bit\ (Rep\text{-}uint\ w)\ (nat\text{-}of\text{-}integer\ n)\ b$)
including *undefined-transfer integer.lifting* ⟨proof⟩

code-printing constant *uint-set-bit* \rightarrow
 (SML) *Uint.set'-bit* **and**
 (Eval) *(raise (Fail Machine dependent code))* **and**
 (Quickcheck) *Uint.set'-bit* **and**
 (Haskell) *Data'-Bits.setBitBounded* **and**
 (OCaml) *Uint.set'-bit* **and**
 (Scala) *Uint.set'-bit*

definition *uint-shiffl* $::\ uint \Rightarrow integer \Rightarrow uint$

where [code del]:

$uint\text{-}shiffl\ x\ n = (if\ n < 0 \vee dflt\text{-}size\text{-}integer \leq n\ then\ undefined\ (push\text{-}bit\ ::\ nat$
 $\Rightarrow uint \Rightarrow -)\ x\ n\ else\ push\text{-}bit\ (nat\text{-}of\text{-}integer\ n)\ x)$

lemma *shiffl-uint-code* [code]: $\text{push-bit } n \ x = (\text{if } n < \text{dflt-size} \text{ then } \text{uint-shiffl } x \text{ (integer-of-nat } n) \text{ else } 0)$

including *undefined-transfer integer.lifting* ⟨proof⟩

lemma *uint-shiffl-code* [code]:

$\text{Rep-uint } (\text{uint-shiffl } w \ n) =$
 $(\text{if } n < 0 \vee \text{dflt-size-integer} \leq n \text{ then } \text{Rep-uint } (\text{undefined } (\text{push-bit} :: \text{nat} \Rightarrow \text{uint}$
 $\Rightarrow -) \ w \ n) \text{ else } \text{push-bit } (\text{nat-of-integer } n) \ (\text{Rep-uint } w))$

including *undefined-transfer integer.lifting* ⟨proof⟩

code-printing constant *uint-shiffl* \rightarrow

(SML) *Uint.shiffl* **and**
 (Eval) (*raise (Fail Machine dependent code)*) **and**
 (Quickcheck) *Uint.shiffl* **and**
 (Haskell) *Data'-Bits.shifflBounded* **and**
 (OCaml) *Uint.shiffl* **and**
 (Scala) *Uint.shiffl*

definition *uint-shiftr* :: $\text{uint} \Rightarrow \text{integer} \Rightarrow \text{uint}$

where [code del]:

$\text{uint-shiftr } x \ n = (\text{if } n < 0 \vee \text{dflt-size-integer} \leq n \text{ then } \text{undefined } (\text{drop-bit} :: \text{nat}$
 $\Rightarrow \text{uint} \Rightarrow -) \ x \ n \text{ else } \text{drop-bit } (\text{nat-of-integer } n) \ x)$

lemma *shiftr-uint-code* [code]: $\text{drop-bit } n \ x = (\text{if } n < \text{dflt-size} \text{ then } \text{uint-shiftr } x \text{ (integer-of-nat } n) \text{ else } 0)$

including *undefined-transfer integer.lifting* ⟨proof⟩

lemma *uint-shiftr-code* [code]:

$\text{Rep-uint } (\text{uint-shiftr } w \ n) =$
 $(\text{if } n < 0 \vee \text{dflt-size-integer} \leq n \text{ then } \text{Rep-uint } (\text{undefined } (\text{drop-bit} :: \text{nat} \Rightarrow \text{uint}$
 $\Rightarrow -) \ w \ n) \text{ else } \text{drop-bit } (\text{nat-of-integer } n) \ (\text{Rep-uint } w))$

including *undefined-transfer integer.lifting* ⟨proof⟩

code-printing constant *uint-shiftr* \rightarrow

(SML) *Uint.shiftr* **and**
 (Eval) (*raise (Fail Machine dependent code)*) **and**
 (Quickcheck) *Uint.shiftr* **and**
 (Haskell) *Data'-Bits.shiftrBounded* **and**
 (OCaml) *Uint.shiftr* **and**
 (Scala) *Uint.shiftr*

definition *uint-sshiftr* :: $\text{uint} \Rightarrow \text{integer} \Rightarrow \text{uint}$

where [code del]:

$\text{uint-sshiftr } x \ n =$
 $(\text{if } n < 0 \vee \text{dflt-size-integer} \leq n \text{ then } \text{undefined } \text{signed-drop-bit-uint } n \ x \text{ else}$
 $\text{signed-drop-bit-uint } (\text{nat-of-integer } n) \ x)$

lemma *sshiftr-uint-code* [code]:

$\text{signed-drop-bit-uint } n \ x =$

(if $n < \text{dflt-size}$ then $\text{uint-sshiftr } x$ ($\text{integer-of-nat } n$) else
 if bit x wivs-index then -1 else 0)

including *undefined-transfer integer.lifting* $\langle \text{proof} \rangle$

lemma *uint-sshiftr-code* [code]:

$\text{Rep-uint } (\text{uint-sshiftr } w \ n) =$

(if $n < 0 \vee \text{dflt-size-integer} \leq n$ then $\text{Rep-uint } (\text{undefined signed-drop-bit-uint } n$
 $w)$ else $\text{signed-drop-bit } (\text{nat-of-integer } n)$ ($\text{Rep-uint } w$))

including *undefined-transfer integer.lifting* $\langle \text{proof} \rangle$

code-printing constant *uint-sshiftr* \rightarrow

(SML) *Uint.shiftr'-signed and*

(Eval) (*raise (Fail Machine dependent code)*) **and**

(Quickcheck) *Uint.shiftr'-signed and*

(Haskell)

(*Prelude.fromInteger (Prelude.toInteger (Data'-Bits.shiftrBounded (Prelude.fromInteger*
Prelude.toInteger -) :: Uint.Int) -) :: Uint.Word) **and**

(OCaml) *Uint.shiftr'-signed and*

(Scala) *Uint.shiftr'-signed*

lemma *uint-msb-test-bit*: $\text{msb } x \longleftrightarrow \text{bit } (x :: \text{uint}) \ \text{wivs-index}$
 $\langle \text{proof} \rangle$

lemma *msb-uint-code* [code]: $\text{msb } x \longleftrightarrow \text{uint-test-bit } x \ \text{wivs-index-integer}$
 $\langle \text{proof} \rangle$

lemma *uint-of-int-code* [code]: $\text{uint-of-int } i = (\text{BITS } n. \text{bit } i \ n)$
 $\langle \text{proof} \rangle$

9.3 Quickcheck setup

definition *uint-of-natural* :: $\text{natural} \Rightarrow \text{uint}$

where *uint-of-natural* $x \equiv \text{Uint } (\text{integer-of-natural } x)$

instantiation *uint* :: {*random, exhaustive, full-exhaustive*} **begin**

definition *random-uint* $\equiv \text{qc-random-cnv } \text{uint-of-natural}$

definition *exhaustive-uint* $\equiv \text{qc-exhaustive-cnv } \text{uint-of-natural}$

definition *full-exhaustive-uint* $\equiv \text{qc-full-exhaustive-cnv } \text{uint-of-natural}$

instance $\langle \text{proof} \rangle$

end

instantiation *uint* :: *narrowing* **begin**

interpretation *quickcheck-narrowing-samples*

$\lambda i. (\text{Uint } i, \text{Uint } (- i)) \ 0$

Typerep.Typerep (*STR "Uint.uint"*) [] $\langle \text{proof} \rangle$

definition *narrowing-uint* $d = \text{qc-narrowing-drawn-from } (\text{narrowing-samples } d) \ d$

declare [[code drop: *partial-term-of* :: *uint itself* \Rightarrow -]]


```
lemmas partial-term-of-uint [code] = partial-term-of-code
```

```
instance <proof>
```

```
end
```

```
end
```


Chapter 10

Conversions between unsigned words and between char

theory *Native-Cast*

imports

Uint8

Uint16

Uint32

Uint64

begin

Auxiliary stuff

lemma *integer-of-char-char-of-integer* [*simp*]:
integer-of-char (char-of-integer x) = x mod 256
<proof>

lemma *char-of-integer-integer-of-char* [*simp*]:
char-of-integer (integer-of-char x) = x
<proof>

lemma *int-lt-numeral* [*simp*]: *int x < numeral n \longleftrightarrow x < numeral n*
<proof>

lemma *int-of-integer-ge-0*: *0 \leq int-of-integer x \longleftrightarrow 0 \leq x*
including *integer.lifting* *<proof>*

lemma *integer-of-char-ge-0* [*simp*]: *0 \leq integer-of-char x*
including *integer.lifting* *<proof>*

10.1 Conversion between native words

lift-definition *wint8-of-uint16* :: *uint16 \Rightarrow uint8* **is** *ucast* *<proof>*

lift-definition *wint8-of-wint32* :: *wint32* ⇒ *wint8* **is** *ucast* ⟨*proof*⟩

lift-definition *wint8-of-wint64* :: *wint64* ⇒ *wint8* **is** *ucast* ⟨*proof*⟩

lift-definition *wint16-of-wint8* :: *wint8* ⇒ *wint16* **is** *ucast* ⟨*proof*⟩

lift-definition *wint16-of-wint32* :: *wint32* ⇒ *wint16* **is** *ucast* ⟨*proof*⟩

lift-definition *wint16-of-wint64* :: *wint64* ⇒ *wint16* **is** *ucast* ⟨*proof*⟩

lift-definition *wint32-of-wint8* :: *wint8* ⇒ *wint32* **is** *ucast* ⟨*proof*⟩

lift-definition *wint32-of-wint16* :: *wint16* ⇒ *wint32* **is** *ucast* ⟨*proof*⟩

lift-definition *wint32-of-wint64* :: *wint64* ⇒ *wint32* **is** *ucast* ⟨*proof*⟩

lift-definition *wint64-of-wint8* :: *wint8* ⇒ *wint64* **is** *ucast* ⟨*proof*⟩

lift-definition *wint64-of-wint16* :: *wint16* ⇒ *wint64* **is** *ucast* ⟨*proof*⟩

lift-definition *wint64-of-wint32* :: *wint32* ⇒ *wint64* **is** *ucast* ⟨*proof*⟩

context

begin

qualified definition *mask* :: *integer*

where ⟨*mask* = (*0xFFFFFFFF* :: *integer*)⟩

end

code-printing

constant *wint8-of-wint16* →

(*SML-word*) *Word8.fromLarge* (*Word16.toLarge* -) **and**

(*Haskell*) (*Prelude.fromIntegral* - :: *Uint8.Word8*) **and**

(*Scala*) -.toByte

| **constant** *wint8-of-wint32* →

(*SML*) *Word8.fromLarge* (*Word32.toLarge* -) **and**

(*Haskell*) (*Prelude.fromIntegral* - :: *Uint8.Word8*) **and**

(*Scala*) -.toByte

| **constant** *wint8-of-wint64* →

(*SML*) *Word8.fromLarge* (*Uint64.toLarge* -) **and**

(*Haskell*) (*Prelude.fromIntegral* - :: *Uint8.Word8*) **and**

(*Scala*) -.toByte

| **constant** *wint16-of-wint8* →

(*SML-word*) *Word16.fromLarge* (*Word8.toLarge* -) **and**

(*Haskell*) (*Prelude.fromIntegral* - :: *Uint16.Word16*) **and**

(*Scala*) ((-).toInt & 0xFF).toChar

| **constant** *wint16-of-wint32* →

(*SML-word*) *Word16.fromLarge* (*Word32.toLarge* -) **and**

(*Haskell*) (*Prelude.fromIntegral* - :: *Uint16.Word16*) **and**

(*Scala*) -.toChar

| **constant** *wint16-of-wint64* →

(*SML-word*) *Word16.fromLarge* (*Uint64.toLarge* -) **and**

(*Haskell*) (*Prelude.fromIntegral* - :: *Uint16.Word16*) **and**

(*Scala*) -.toChar

| **constant** *wint32-of-wint8* →

```

(SML) Word32.fromLarge (Word8.toLarge -) and
(Haskell) (Prelude.fromIntegral - :: Uint32.Word32) and
(Scala) ((-).toInt & 0xFF)
| constant uint32-of-uint16  $\rightarrow$ 
(SML-word) Word32.fromLarge (Word16.toLarge -) and
(Haskell) (Prelude.fromIntegral - :: Uint32.Word32) and
(Scala) (-).toInt
| constant uint32-of-uint64  $\rightarrow$ 
(SML-word) Word32.fromLarge (Uint64.toLarge -) and
(Haskell) (Prelude.fromIntegral - :: Uint32.Word32) and
(Scala) (-).toInt and
(OCaml) Int64.to'-int32
| constant uint64-of-uint8  $\rightarrow$ 
(SML-word) Word64.fromLarge (Word8.toLarge -) and
(Haskell) (Prelude.fromIntegral - :: Uint64.Word64) and
(Scala) ((-).toLong & 0xFF)
| constant uint64-of-uint16  $\rightarrow$ 
(SML-word) Word64.fromLarge (Word16.toLarge -) and
(Haskell) (Prelude.fromIntegral - :: Uint64.Word64) and
(Scala) -.toLong
| constant uint64-of-uint32  $\rightarrow$ 
(SML-word) Word64.fromLarge (Word32.toLarge -) and
(Haskell) (Prelude.fromIntegral - :: Uint64.Word64) and
(Scala) ((-).toLong & 0xFFFFFFFFL) and
(OCaml) Int64.logand (Int64.of'-int32 -) (Int64.of'-string 4294967295)

```

Use *Abs-uint8'* etc. instead of *Rep-uint8* in code equations for conversion functions to avoid exceptions during code generation when the target language provides only some of the uint types.

```

lemma uint8-of-uint16-code [code]:
  uint8-of-uint16 x = Abs-uint8' (ucast (Rep-uint16' x))
<proof>

```

```

lemma uint8-of-uint32-code [code]:
  uint8-of-uint32 x = Abs-uint8' (ucast (Rep-uint32' x))
<proof>

```

```

lemma uint8-of-uint64-code [code]:
  uint8-of-uint64 x = Abs-uint8' (ucast (Rep-uint64' x))
<proof>

```

```

lemma uint16-of-uint8-code [code]:
  uint16-of-uint8 x = Abs-uint16' (ucast (Rep-uint8' x))
<proof>

```

```

lemma uint16-of-uint32-code [code]:
  uint16-of-uint32 x = Abs-uint16' (ucast (Rep-uint32' x))
<proof>

```

lemma *uint16-of-uint64-code* [code]:
 $\text{uint16-of-uint64 } x = \text{Abs-uint16}' (\text{ucast } (\text{Rep-uint64}' x))$
 ⟨proof⟩

lemma *uint32-of-uint8-code* [code]:
 $\text{uint32-of-uint8 } x = \text{Abs-uint32}' (\text{ucast } (\text{Rep-uint8}' x))$
 ⟨proof⟩

lemma *uint32-of-uint16-code* [code]:
 $\text{uint32-of-uint16 } x = \text{Abs-uint32}' (\text{ucast } (\text{Rep-uint16}' x))$
 ⟨proof⟩

lemma *uint32-of-uint64-code* [code]:
 $\text{uint32-of-uint64 } x = \text{Abs-uint32}' (\text{ucast } (\text{Rep-uint64}' x))$
 ⟨proof⟩

lemma *uint64-of-uint8-code* [code]:
 $\text{uint64-of-uint8 } x = \text{Abs-uint64}' (\text{ucast } (\text{Rep-uint8}' x))$
 ⟨proof⟩

lemma *uint64-of-uint16-code* [code]:
 $\text{uint64-of-uint16 } x = \text{Abs-uint64}' (\text{ucast } (\text{Rep-uint16}' x))$
 ⟨proof⟩

lemma *uint64-of-uint32-code* [code]:
 $\text{uint64-of-uint32 } x = \text{Abs-uint64}' (\text{ucast } (\text{Rep-uint32}' x))$
 ⟨proof⟩

end

theory *Native-Cast-Uint* **imports**

Native-Cast

Uint

begin

lift-definition *uint-of-uint8* :: *uint8* ⇒ *uint* **is** *ucast* ⟨proof⟩

lift-definition *uint-of-uint16* :: *uint16* ⇒ *uint* **is** *ucast* ⟨proof⟩

lift-definition *uint-of-uint32* :: *uint32* ⇒ *uint* **is** *ucast* ⟨proof⟩

lift-definition *uint-of-uint64* :: *uint64* ⇒ *uint* **is** *ucast* ⟨proof⟩

lift-definition *uint8-of-uint* :: *uint* ⇒ *uint8* **is** *ucast* ⟨proof⟩

lift-definition *uint16-of-uint* :: *uint* ⇒ *uint16* **is** *ucast* ⟨proof⟩

lift-definition *uint32-of-uint* :: *uint* ⇒ *uint32* **is** *ucast* ⟨proof⟩

lift-definition *uint64-of-uint* :: *uint* ⇒ *uint64* **is** *ucast* ⟨proof⟩

code-printing

constant *uint-of-uint8* ←

(*SML*) *Word.fromLarge* (*Word8.toLarge* -) **and**

```

(Haskell) (Prelude.fromIntegral - :: Uint.Word) and
(Scala) ((-).toInt & 0xFF)
| constant uint-of-uint16 →
(SML-word) Word.fromLarge (Word16.toLarge -) and
(Haskell) (Prelude.fromIntegral - :: Uint.Word) and
(Scala) (-).toInt
| constant uint-of-uint32 →
(SML) Word.fromLarge (Word32.toLarge -) and
(Haskell) (Prelude.fromIntegral - :: Uint.Word) and
(Scala) - and
(OCaml) (Int32.to'-int -) land Uint.int'-mask
| constant uint-of-uint64 →
(SML) Word.fromLarge (Uint64.toLarge -) and
(Haskell) (Prelude.fromIntegral - :: Uint.Word) and
(Scala) (-).toInt and
(OCaml) Int64.to'-int
| constant uint8-of-uint →
(SML) Word8.fromLarge (Word.toLarge -) and
(Haskell) (Prelude.fromIntegral - :: Uint8.Word8) and
(Scala) (-).toByte
| constant uint16-of-uint →
(SML-word) Word16.fromLarge (Word.toLarge -) and
(Haskell) (Prelude.fromIntegral - :: Uint16.Word16) and
(Scala) (-).toChar
| constant uint32-of-uint →
(SML) Word32.fromLarge (Word.toLarge -) and
(Haskell) (Prelude.fromIntegral - :: Uint32.Word32) and
(Scala) - and
(OCaml) Int32.logand (Int32.of'-int -) Uint.int32'-mask
| constant uint64-of-uint →
(SML) Uint64.fromLarge (Word.toLarge -) and
(Haskell) (Prelude.fromIntegral - :: Uint64.Word64) and
(Scala) ((-).toLong & 0xFFFFFFFFL) and
(OCaml) Int64.logand (Int64.of'-int -) Uint.int64'-mask

```

```

lemma uint8-of-uint-code [code]:
  uint8-of-uint x = Abs-uint8' (ucast (Rep-uint' x))
  ⟨proof⟩

```

```

lemma uint16-of-uint-code [code]:
  uint16-of-uint x = Abs-uint16' (ucast (Rep-uint' x))
  ⟨proof⟩

```

```

lemma uint32-of-uint-code [code]:
  uint32-of-uint x = Abs-uint32' (ucast (Rep-uint' x))
  ⟨proof⟩

```

```

lemma uint64-of-uint-code [code]:
  uint64-of-uint x = Abs-uint64' (ucast (Rep-uint' x))

```

<proof>

lemma *wint-of-wint8-code* [code]:

wint-of-wint8 $x = \text{Abs-wint}' (\text{ucast} (\text{Rep-wint8}' x))$

<proof>

lemma *wint-of-wint16-code* [code]:

wint-of-wint16 $x = \text{Abs-wint}' (\text{ucast} (\text{Rep-wint16}' x))$

<proof>

lemma *wint-of-wint32-code* [code]:

wint-of-wint32 $x = \text{Abs-wint}' (\text{ucast} (\text{Rep-wint32}' x))$

<proof>

lemma *wint-of-wint64-code* [code]:

wint-of-wint64 $x = \text{Abs-wint}' (\text{ucast} (\text{Rep-wint64}' x))$

<proof>

end

10.2 Compatibility with Imperative/HOL

theory *Native-Word-Imperative-HOL* **imports**

Code-Target-Word-Base

HOL-Imperative-HOL.Heap-Monad

begin

We add a code target that combines the translations for native words that are by default not supported by all PolyML versions with the adaptations for `Imperative_HOL`.

<ML>

end

Chapter 11

Test cases

theory *Native-Word-Test* **imports**

Uint64 Uint32 Uint16 Uint8 Uint Native-Cast-Uint

HOL-Library.Code-Test Word-Lib.Bit-Shifts-Infix-Syntax

begin

export-code

nat-of-uint8 uint8-of-nat

nat-of-uint16 uint16-of-nat

nat-of-uint32 uint32-of-nat

nat-of-uint64 uint64-of-nat

nat-of-uint uint-of-nat

in *SML*

11.1 Tests for *uint32*

context

includes *bit-operations-syntax*

begin

abbreviation (*input*) *sshiftr-uint32* (**infixl** $\gg\gg$ 55)

where $\langle w \gg\gg n \equiv \text{signed-drop-bit-uint32 } n \ w \rangle$

definition *test-uint32* **where**

test-uint32 \longleftrightarrow

$(([$ *0x100000001*, -1 , -4294967291 , *0xFFFFFFFF*, *0x12345678*

, *0x5A AND 0x36*

, *0x5A OR 0x36*

, *0x5A XOR 0x36*

, *NOT 0x5A*

, $5 + 6$, $-5 + 6$, $-6 + 5$, $-5 + (-6)$, $0xFFFFFFFF + 1$

, $5 - 3$, $3 - 5$

, $5 * 3$, $-5 * 3$, $-5 * -4$, $0x12345678 * 0x87654321$

, $5 \text{ div } 3$, $-5 \text{ div } 3$, $-5 \text{ div } -3$, $5 \text{ div } -3$

, $5 \text{ mod } 3$, $-5 \text{ mod } 3$, $-5 \text{ mod } -3$, $5 \text{ mod } -3$

```

, set-bit 5 4 True, set-bit (- 5) 2 True, set-bit 5 0 False, set-bit (- 5) 1 False
, set-bit 5 32 True, set-bit 5 32 False, set-bit (- 5) 32 True, set-bit (- 5) 32
False
, 1 << 2, -1 << 3, 1 << 32, 1 << 0
, 100 >> 3, -100 >> 3, 100 >> 32, -100 >> 32
, 100 >>> 3, -100 >>> 3, 100 >>> 32, -100 >>> 32] :: uint32 list)
=
[ 1, 4294967295, 5, 4294967295, 305419896
, 18
, 126
, 108
, 4294967205
, 11, 1, 4294967295, 4294967285, 0
, 2, 4294967294
, 15, 4294967281, 20, 1891143032
, 1, 1431655763, 0, 0
, 2, 2, 4294967291, 5
, 21, 4294967295, 4, 4294967289
, 5, 5, 4294967291, 4294967291
, 4, 4294967288, 0, 1
, 12, 536870899, 0, 0
, 12, 4294967283, 0, 4294967295]) ^
([ (0x5 :: uint32) = 0x5, (0x5 :: uint32) = 0x6
, (0x5 :: uint32) < 0x5, (0x5 :: uint32) < 0x6, (-5 :: uint32) < 6, (6 :: uint32)
< -5
, (0x5 :: uint32) ≤ 0x5, (0x5 :: uint32) ≤ 0x4, (-5 :: uint32) ≤ 6, (6 :: uint32)
≤ -5
, (0x7FFFFFFF :: uint32) < 0x80000000, (0xFFFFFFFF :: uint32) < 0,
(0x80000000 :: uint32) < 0x7FFFFFFF
, bit (0x7FFFFFFF :: uint32) 0, bit (0x7FFFFFFF :: uint32) 31, bit (0x80000000
:: uint32) 31, bit (0x80000000 :: uint32) 32
]
) ^
([ integer-of-uint32 0, integer-of-uint32 0x7FFFFFFF, integer-of-uint32 0x80000000,
integer-of-uint32 0xAAAAAAAAA]
)
=
[0, 0x7FFFFFFF, 0x80000000, 0xAAAAAAAAA]

```

export-code test-uint32 checking SML Haskell? OCaml? Scala

notepad begin

<proof>

end

definition *test-uint32'* :: *uint32*
where *test-uint32'* = 0 + 10 - 14 * 3 div 6 mod 3 << 3 >> 2
 ⟨ML⟩

lemma *x AND y = x OR (y :: uint32)*
quickcheck[*random, expect=counterexample*]
quickcheck[*exhaustive, expect=counterexample*]
 ⟨proof⟩

lemma (*x :: uint32*) *AND x = x OR x*
quickcheck[*narrowing, expect=no-counterexample*]
 ⟨proof⟩

lemma (*f :: uint32 ⇒ unit*) = *g*
quickcheck[*narrowing, size=3, expect=no-counterexample*]
 ⟨proof⟩

end

11.2 Tests for *uint16*

context
includes *bit-operations-syntax*
begin

abbreviation (*input*) *sshiftr-uint16* (**infixl** >>> 55)
where ⟨*w* >>> *n* ≡ *signed-drop-bit-uint16 n w*⟩

definition *test-uint16* **where**
test-uint16 ↔
 (([*0x10001*, -1, -65535, *0xFFFF*, *0x1234*
 , *0x5A AND 0x36*
 , *0x5A OR 0x36*
 , *0x5A XOR 0x36*
 , *NOT 0x5A*
 , *5 + 6*, -5 + 6, -6 + 5, -5 + -6, *0xFFFF + 1*
 , *5 - 3*, *3 - 5*
 , *5 * 3*, -5 * 3, -5 * -4, *0x1234 * 0x8765*
 , *5 div 3*, -5 div 3, -5 div -3, *5 div -3*
 , *5 mod 3*, -5 mod 3, -5 mod -3, *5 mod -3*
 , *set-bit 5 4 True*, *set-bit (- 5) 2 True*, *set-bit 5 0 False*, *set-bit (- 5) 1 False*
 , *set-bit 5 32 True*, *set-bit 5 32 False*, *set-bit (- 5) 32 True*, *set-bit (- 5) 32*
False
 , *1 << 2*, -1 << 3, *1 << 16*, *1 << 0*
 , *100 >> 3*, -100 >> 3, *100 >> 16*, -100 >> 16
 , *100 >>> 3*, -100 >>> 3, *100 >>> 16*, -100 >>> 16] :: *uint16 list*)
 =
 [1, 65535, 1, 65535, 4660

```

    , 18
    , 126
    , 108
    , 65445
    , 11, 1, 65535, 65525, 0
    , 2, 65534
    , 15, 65521, 20, 39556
    , 1, 21843, 0, 0
    , 2, 2, 65531, 5
    , 21, 65535, 4, 65529
    , 5, 5, 65531, 65531
    , 4, 65528, 0, 1
    , 12, 8179, 0, 0
    , 12, 65523, 0, 65535]) ^
  ([ (0x5 :: uint16) = 0x5, (0x5 :: uint16) = 0x6
    , (0x5 :: uint16) < 0x5, (0x5 :: uint16) < 0x6, (-5 :: uint16) < 6, (6 :: uint16)
< -5
    , (0x5 :: uint16) ≤ 0x5, (0x5 :: uint16) ≤ 0x4, (-5 :: uint16) ≤ 6, (6 :: uint16)
≤ -5
    , (0x7FFF :: uint16) < 0x8000, (0xFFFF :: uint16) < 0, (0x8000 :: uint16) <
0x7FFF
    , bit (0x7FFF :: uint16) 0, bit (0x7FFF :: uint16) 15, bit (0x8000 :: uint16)
15, bit (0x8000 :: uint16) 16
    ]
  =
  [ True, False
    , False, True, False, True
    , True, False, False, True
    , True, False, False
    , True, False, True, False
  ] ^
  ([integer-of-uint16 0, integer-of-uint16 0x7FFF, integer-of-uint16 0x8000, inte-
ger-of-uint16 0xAAAA]
  =
  [0, 0x7FFF, 0x8000, 0xAAAA])

```

export-code *test-uint16 checking Haskell? Scala*

export-code *test-uint16 checking SML-word*

notepad begin

<proof>

end

lemma *(x :: uint16) AND x = x OR x*

quickcheck*[narrowing, expect=no-counterexample]*

<proof>

lemma *(f :: uint16 ⇒ unit) = g*

quickcheck*[narrowing, size=3, expect=no-counterexample]*

<proof>

end

11.3 Tests for *uint8*

context

includes *bit-operations-syntax*

begin

abbreviation (*input*) *sshiftr-uint8* (**infixl** >>> 55)

where $\langle w \gg \rangle n \equiv \text{signed-drop-bit-uint8 } n \ w$

definition *test-uint8* **where**

test-uint8 \longleftrightarrow

(([*0x101*, *-1*, *-255*, *0xFF*, *0x12*
, *0x5A AND 0x36*
, *0x5A OR 0x36*
, *0x5A XOR 0x36*
, *NOT 0x5A*
, *5 + 6*, *-5 + 6*, *-6 + 5*, *-5 + -6*, *0xFF + 1*
, *5 - 3*, *3 - 5*
, *5 * 3*, *-5 * 3*, *-5 * -4*, *0x12 * 0x87*
, *5 div 3*, *-5 div 3*, *-5 div -3*, *5 div -3*
, *5 mod 3*, *-5 mod 3*, *-5 mod -3*, *5 mod -3*
, *set-bit 5 4 True*, *set-bit (- 5) 2 True*, *set-bit 5 0 False*, *set-bit (- 5) 1 False*
, *set-bit 5 32 True*, *set-bit 5 32 False*, *set-bit (- 5) 32 True*, *set-bit (- 5) 32*

False

, *1 << 2*, *-1 << 3*, *1 << 8*, *1 << 0*
, *100 >> 3*, *-100 >> 3*, *100 >> 8*, *-100 >> 8*
, *100 >>> 3*, *-100 >>> 3*, *100 >>> 8*, *-100 >>> 8*] :: *uint8 list*)

=

[*1*, *255*, *1*, *255*, *18*
, *18*
, *126*
, *108*
, *165*
, *11*, *1*, *255*, *245*, *0*
, *2*, *254*
, *15*, *241*, *20*, *126*
, *1*, *83*, *0*, *0*
, *2*, *2*, *251*, *5*
, *21*, *255*, *4*, *249*
, *5*, *5*, *251*, *251*
, *4*, *248*, *0*, *1*
, *12*, *19*, *0*, *0*
, *12*, *243*, *0*, *255*] \wedge

([(*0x5 :: uint8*) = *0x5*, (*0x5 :: uint8*) = *0x6*
, (*0x5 :: uint8*) < *0x5*, (*0x5 :: uint8*) < *0x6*, (*-5 :: uint8*) < *6*, (*6 :: uint8*) <

```

-5
, (0x5 :: uint8) ≤ 0x5, (0x5 :: uint8) ≤ 0x4, (-5 :: uint8) ≤ 6, (6 :: uint8) ≤
-5
, (0x7F :: uint8) < 0x80, (0xFF :: uint8) < 0, (0x80 :: uint8) < 0x7F
, bit (0x7F :: uint8) 0, bit (0x7F :: uint8) 7, bit (0x80 :: uint8) 7, bit (0x80 ::
uint8) 8
]
=
[ True, False
, False, True, False, True
, True, False, False, True
, True, False, False
, True, False, True, False
] ^
(integer-of-uint8 0, integer-of-uint8 0x7F, integer-of-uint8 0x80, integer-of-uint8
0xAA)
=
[0, 0x7F, 0x80, 0xAA]

```

export-code *test-uint8* **checking** *SML Haskell? Scala*

notepad **begin**

```

⟨proof⟩
end
⟨ML⟩

```

definition *test-uint8'* :: *uint8*

```

where test-uint8' = 0 + 10 - 14 * 3 div 6 mod 3 << 3 >> 2
⟨ML⟩

```

lemma *x AND y = x OR (y :: uint8)*

```

quickcheck[random, expect=counterexample]
quickcheck[exhaustive, expect=counterexample]
⟨proof⟩

```

lemma (*x :: uint8*) *AND x = x OR x*

```

quickcheck[narrowing, expect=no-counterexample]
⟨proof⟩

```

lemma (*f :: uint8 ⇒ unit*) = *g*

```

quickcheck[narrowing, size=3, expect=no-counterexample]
⟨proof⟩

```

11.4 Tests for *uint*

context

includes *bit-operations-syntax*

begin

abbreviation (*input*) *sshiftr-uint* (**infixl** >>> 55)
where $\langle w \gg \rangle n \equiv \text{signed-drop-bit-uint } n \ w$

definition *test-uint* \equiv *let*

```

test-list1 = (let
  HS = uint-of-int ( $2 \wedge (\text{dfft-size} - 1)$ )
  in
  ([ HS + HS + 1, -1, -HS - HS + 5, HS + (HS - 1), 0x12
    , 0x5A AND 0x36
    , 0x5A OR 0x36
    , 0x5A XOR 0x36
    , NOT 0x5A
    , 5 + 6, -5 + 6, -6 + 5, -5 + -6, HS + (HS - 1) + 1
    , 5 - 3, 3 - 5
    , 5 * 3, -5 * 3, -5 * -4, 0x12345678 * 0x87654321]
  @ (if dfft-size > 4 then
    [ 5 div 3, -5 div 3, -5 div -3, 5 div -3
      , 5 mod 3, -5 mod 3, -5 mod -3, 5 mod -3
      , set-bit 5 4 True, set-bit (- 5) 2 True, set-bit 5 0 False, set-bit (- 5) 1 False
      , set-bit 5 dfft-size True, set-bit 5 dfft-size False, set-bit (- 5) dfft-size True,
    set-bit (- 5) dfft-size False
      , 1 << 2, -1 << 3, push-bit dfft-size 1, 1 << 0
      , 31 >> 3, -1 >> 3, 31 >> dfft-size, -1 >> dfft-size
      , 15 >>> 2, -1 >>> 3, 15 >>> dfft-size, -1 >>> dfft-size]
    else []) :: uint list));
```

test-list2 = (*let*

```

  S = wivs-shift
  in
  ([ 1, -1, -S + 5, S - 1, 0x12
    , 0x5A AND 0x36
    , 0x5A OR 0x36
    , 0x5A XOR 0x36
    , NOT 0x5A
    , 5 + 6, -5 + 6, -6 + 5, -5 + -6, 0
    , 5 - 3, 3 - 5
    , 5 * 3, -5 * 3, -5 * -4, 0x12345678 * 0x87654321]
  @ (if dfft-size > 4 then
    [ 5 div 3, (S - 5) div 3, (S - 5) div (S - 3), 5 div (S - 3)
      , 5 mod 3, (S - 5) mod 3, (S - 5) mod (S - 3), 5 mod (S - 3)
      , set-bit 5 4 True, -1, set-bit 5 0 False, -7
      , 5, 5, -5, -5
      , 4, -8, 0, 1
      , 3, (S >> 3) - 1, 0, 0
      , 3, (S >> 1) + (S >> 1) - 1, 0, -1]
    else []) :: int list));
```

test-list-c1 = (*let*

```

    HS = uint-of-int ((2^(dflt-size - 1)))
  in
  [ (0x5 :: uint) = 0x5, (0x5 :: uint) = 0x6
  , (0x5 :: uint) < 0x5, (0x5 :: uint) < 0x6, (-5 :: uint) < 6, (6 :: uint) < -5
  , (0x5 :: uint) ≤ 0x5, (0x5 :: uint) ≤ 0x4, (-5 :: uint) ≤ 6, (6 :: uint) ≤ -5
  , (HS - 1) < HS, (HS + HS - 1) < 0, HS < HS - 1
  , bit (HS - 1) 0, bit (HS - 1 :: uint) (dflt-size - 1), bit (HS :: uint) (dflt-size
- 1), bit (HS :: uint) dflt-size
  ]];

test-list-c2 =
  [ True, False
  , False, dflt-size ≥ 2, dflt-size = 3, dflt-size ≠ 3
  , True, False, dflt-size = 3, dflt-size ≠ 3
  , True, False, False
  , dflt-size ≠ 1, False, True, False
  ]
in
  test-list1 = map uint-of-int test-list2
  ∧ test-list-c1 = test-list-c2

```

export-code *test-uint checking SML Haskell? OCaml? Scala*

lemma *test-uint*

quickcheck[*exhaustive, expect=no-counterexample*]
 ⟨*proof*⟩

lemma *x AND y = x OR (y :: uint)*

quickcheck[*random, expect=counterexample*]
quickcheck[*exhaustive, expect=counterexample*]
 ⟨*proof*⟩

lemma *(x :: uint) AND x = x OR x*

quickcheck[*narrowing, expect=no-counterexample*]
 ⟨*proof*⟩

lemma *(f :: uint ⇒ unit) = g*

quickcheck[*narrowing, size=3, expect=no-counterexample*]
 ⟨*proof*⟩

11.5 Tests for *uint64*

context

includes *bit-operations-syntax*

begin

abbreviation *(input) sshiftr-uint64 (infixl >>> 55)*

where ⟨*w >>> n* ≡ *signed-drop-bit-uint64 n w*⟩

definition *test-uint64* where

```

test-uint64  $\longleftrightarrow$ 
([ 0x10000000000000001, -1, -9223372036854775808, 0xFFFFFFFFFFFFFFFF,
0x1234567890ABCDEF
, 0x5A AND 0x36
, 0x5A OR 0x36
, 0x5A XOR 0x36
, NOT 0x5A
, 5 + 6, -5 + 6, -6 + 5, -5 + (- 6), 0xFFFFFFFFFFFFFFFF + 1
, 5 - 3, 3 - 5
, 5 * 3, -5 * 3, -5 * -4, 0x1234567890ABCDEF * 0xFEDCBA0987654321
, 5 div 3, -5 div 3, -5 div -3, 5 div -3
, 5 mod 3, -5 mod 3, -5 mod -3, 5 mod -3
, set-bit 5 4 True, set-bit (- 5) 2 True, set-bit 5 0 False, set-bit (- 5) 1 False
, set-bit 5 64 True, set-bit 5 64 False, set-bit (- 5) 64 True, set-bit (- 5) 64
False
, 1 << 2, -1 << 3, 1 << 64, 1 << 0
, 100 >> 3, -100 >> 3, 100 >> 64, -100 >> 64
, 100 >>> 3, -100 >>> 3, 100 >>> 64, -100 >>> 64] :: uint64 list)
=
[ 1, 18446744073709551615, 9223372036854775808, 18446744073709551615,
1311768467294899695
, 18
, 126
, 108
, 18446744073709551525
, 11, 1, 18446744073709551615, 18446744073709551605, 0
, 2, 18446744073709551614
, 15, 18446744073709551601, 20, 14000077364136384719
, 1, 6148914691236517203, 0, 0
, 2, 2, 18446744073709551611, 5
, 21, 18446744073709551615, 4, 18446744073709551609
, 5, 5, 18446744073709551611, 18446744073709551611
, 4, 18446744073709551608, 0, 1
, 12, 2305843009213693939, 0, 0
, 12, 18446744073709551603, 0, 18446744073709551615]  $\wedge$ 
([ (0x5 :: uint64) = 0x5, (0x5 :: uint64) = 0x6
, (0x5 :: uint64) < 0x5, (0x5 :: uint64) < 0x6, (-5 :: uint64) < 6, (6 :: uint64)
< -5
, (0x5 :: uint64)  $\leq$  0x5, (0x5 :: uint64)  $\leq$  0x4, (-5 :: uint64)  $\leq$  6, (6 :: uint64)
 $\leq$  -5
, (0x7FFFFFFFFFFFFFFFFF :: uint64) < 0x8000000000000000, (0xFFFFFFFFFFFFFFFF
:: uint64) < 0, (0x8000000000000000 :: uint64) < 0x7FFFFFFFFFFFFFFFFF
, bit (0x7FFFFFFFFFFFFFFFFF :: uint64) 0, bit (0x7FFFFFFFFFFFFFFFFF ::
uint64) 63, bit (0x8000000000000000 :: uint64) 63, bit (0x8000000000000000 ::
uint64) 64
]
=
[ True, False

```

```

, False, True, False, True
, True, False, False, True
, True, False, False
, True, False, True, False
]) ^
([integer-of-uint64 0, integer-of-uint64 0x7FFFFFFFFFFFFFFF, integer-of-uint64
0x8000000000000000, integer-of-uint64 0xAAAAAAAAAAAAAAAA]
=
[0, 0x7FFFFFFFFFFFFFFF, 0x8000000000000000, 0xAAAAAAAAAAAAAAAA])

value [nbe] [0x10000000000000001, -1, -9223372036854775808, 0xFFFFFFFFFFFFFFFF,
0x1234567890ABCDEF
, 0x5A AND 0x36
, 0x5A OR 0x36
, 0x5A XOR 0x36
, NOT 0x5A
, 5 + 6, -5 + 6, -6 + 5, -5 + (- 6), 0xFFFFFFFFFFFFFFFF + 1
, 5 - 3, 3 - 5
, 5 * 3, -5 * 3, -5 * -4, 0x1234567890ABCDEF * 0xFEDCBA0987654321
, 5 div 3, -5 div 3, -5 div -3, 5 div -3
, 5 mod 3, -5 mod 3, -5 mod -3, 5 mod -3
, set-bit 5 4 True, set-bit (- 5) 2 True, set-bit 5 0 False, set-bit (- 5) 1 False
, set-bit 5 64 True, set-bit 5 64 False, set-bit (- 5) 64 True, set-bit (- 5) 64
False
, 1 << 2, -1 << 3, 1 << 64, 1 << 0
, 100 >> 3, -100 >> 3, 100 >> 64, -100 >> 64
, 100 >>> 3, -100 >>> 3, 100 >>> 64, -100 >>> 64] :: uint64 list

```

export-code *test-uint64* **checking** *SML Haskell? OCaml? Scala*

notepad begin

<proof>

end

<ML>

definition *test-uint64'* :: *uint64*

where *test-uint64'* = 0 + 10 - 14 * 3 div 6 mod 3 << 3 >> 2

end

11.6 Tests for casts

definition *test-casts* :: *bool*

where *test-casts* \longleftrightarrow

```

map uint8-of-uint32 [10, 0, 0xFE, 0xFFFFFFFF] = [10, 0, 0xFE, 0xFF] ^
map uint8-of-uint64 [10, 0, 0xFE, 0xFFFFFFFFFFFFFFFF] = [10, 0, 0xFE,
0xFF] ^
map uint32-of-uint8 [10, 0, 0xFF] = [10, 0, 0xFF] ^
map uint64-of-uint8 [10, 0, 0xFF] = [10, 0, 0xFF]

```

definition *test-casts'* :: *bool*

where *test-casts'* \longleftrightarrow

map uint8-of-uint16 [10, 0, 0xFE, 0xFFFF] = [10, 0, 0xFE, 0xFF] \wedge
map uint16-of-uint8 [10, 0, 0xFF] = [10, 0, 0xFF] \wedge
map uint16-of-uint32 [10, 0, 0xFFFE, 0xFFFFFFFF] = [10, 0, 0xFFFE, 0xFFFF]
 \wedge
map uint16-of-uint64 [10, 0, 0xFFFE, 0xFFFFFFFFFFFFFFFF] = [10, 0,
 0xFFFE, 0xFFFF] \wedge
map uint32-of-uint16 [10, 0, 0xFFFF] = [10, 0, 0xFFFF] \wedge
map uint64-of-uint16 [10, 0, 0xFFFF] = [10, 0, 0xFFFF]

definition *test-casts''* :: *bool*

where *test-casts''* \longleftrightarrow

map uint32-of-uint64 [10, 0, 0xFFFFFFFFFE, 0xFFFFFFFFFFFFFFFF] = [10,
 0, 0xFFFFFFFFFE, 0xFFFFFFFF] \wedge
map uint64-of-uint32 [10, 0, 0xFFFFFFFF] = [10, 0, 0xFFFFFFFF]

export-code *test-casts test-casts'' checking SML Haskell? Scala*

export-code *test-casts'' checking OCaml?*

export-code *test-casts' checking Haskell? Scala*

notepad begin

<proof>

end

<ML>

definition *test-casts-uint* :: *bool* **where**

test-casts-uint \longleftrightarrow
map uint-of-uint32 ([0, 10] @ (if *dflt-size* < 32 then [push-bit (*dflt-size* - 1) 1,
 0xFFFFFFFF] else [0xFFFFFFFF])) =
 [0, 10] @ (if *dflt-size* < 32 then [push-bit (*dflt-size* - 1) 1, (push-bit *dflt-size* 1)
 - 1] else [0xFFFFFFFF]) \wedge
map uint32-of-uint [0, 10, if *dflt-size* < 32 then push-bit (*dflt-size* - 1) 1 else
 0xFFFFFFFF] =
 [0, 10, if *dflt-size* < 32 then push-bit (*dflt-size* - 1) 1 else 0xFFFFFFFF] \wedge
map uint-of-uint64 [0, 10, push-bit (*dflt-size* - 1) 1, 0xFFFFFFFFFFFFFFFF]
 =
 [0, 10, push-bit (*dflt-size* - 1) 1, (push-bit *dflt-size* 1) - 1] \wedge
map uint64-of-uint [0, 10, push-bit (*dflt-size* - 1) 1] =
 [0, 10, push-bit (*dflt-size* - 1) 1]

definition *test-casts-uint'* :: *bool* **where**

test-casts-uint' \longleftrightarrow
map uint-of-uint16 [0, 10, 0xFFFF] = [0, 10, 0xFFFF] \wedge
map uint16-of-uint [0, 10, 0xFFFF] = [0, 10, 0xFFFF]

definition *test-casts-uint''* :: *bool* **where**

test-casts-uint'' \longleftrightarrow

```
map uint-of-uint8 [0, 10, 0xFF] = [0, 10, 0xFF] ^  
map uint8-of-uint [0, 10, 0xFF] = [0, 10, 0xFF]
```

end

end

end

Chapter 12

Implementation of bit operations on int by target language operations

```
theory Code-Target-Int-Bit
  imports
    Code-Target-Integer-Bit
    HOL-Library.Code-Target-Int
begin

context
  includes bit-operations-syntax
begin

declare [[code drop:
  lsb :: int  $\Rightarrow$  - int-of-integer-symbolic
]]

lemma [code-unfold]:
   $\langle$ of-bool (odd i) = i AND 1 $\rangle$  for i :: int
   $\langle$ proof $\rangle$ 

lemma [code-unfold]:
   $\langle$ bit x n  $\longleftrightarrow$  x AND (push-bit n 1)  $\neq$  0 $\rangle$  for x :: int
   $\langle$ proof $\rangle$ 

context
  includes integer.lifting
begin

lemma lsb-int-code [code]:
  lsb (int-of-integer x) = lsb x
   $\langle$ proof $\rangle$ 
```

lemma *set-bit-int-code* [code]:
 $set-bit (int-of-integer\ x)\ n\ b = int-of-integer (set-bit\ x\ n\ b)$
 ⟨proof⟩

lemma *int-of-integer-symbolic-code* [code]:
 $int-of-integer-symbolic = int-of-integer$
 ⟨proof⟩

context
begin

qualified definition *even* :: $\langle int \Rightarrow bool \rangle$
where [code-abbrev]: $\langle even = Parity.even \rangle$

end

lemma [code]:
 $\langle Code-Target-Int-Bit.even\ i \longleftrightarrow i\ AND\ 1 = 0 \rangle$
 ⟨proof⟩

lemma *bin-rest-code*:
 $int-of-integer\ i\ div\ 2 = int-of-integer (bin-rest-integer\ i)$
 ⟨proof⟩

end

end

end

theory *Native-Word-Test-Emu* **imports**
Native-Word-Test
Code-Target-Int-Bit
begin

12.1 Test cases for emulation of native words

12.1.1 Tests for *uint16*

Test that *uint16* is emulated for PolyML and OCaml via *16 word* if *Native-Word.Code-Target-Int-Bit* is imported.

definition *test-uint16-emulation* :: *bool* **where**
 $test-uint16-emulation \longleftrightarrow (0xFFFF - 0x1000 = (0xEFFF :: uint16))$

export-code *test-uint16-emulation* **checking** *SML OCaml?*
 — test the other target languages as well *Haskell? Scala*

```

notepad begin
<proof>
end

<ML>

lemma x AND y = x OR (y :: uint16)
quickcheck[random, expect=counterexample]
quickcheck[exhaustive, expect=counterexample]
<proof>

```

12.1.2 Tests for *uint8*

Test that *uint8* is emulated for OCaml via *8 word* if *Native-Word.Code-Target-Int-Bit* is imported.

```

definition test-uint8-emulation :: bool where
  test-uint8-emulation  $\longleftrightarrow$  (0xFFF - 0x10 = (0xEF :: uint8))

```

```

export-code test-uint8-emulation checking OCaml?
  — test the other target languages as well SML Haskell? Scala

```

```
end
```

```

theory Native-Word-Test-PolyML imports
  Native-Word-Test
begin

```

12.2 Test with PolyML

```

test-code
  test-uint64 test-uint64' = 0x12
  test-uint32 test-uint32' = 0x12
  test-uint8 test-uint8' = 0x12
  test-uint
  test-casts test-casts''
  test-casts-uint test-casts-uint''
in PolyML

```

```
end
```

```

theory Native-Word-Test-PolyML2 imports
  Native-Word-Test-Emu
begin

```

```
test-code
```

```

    test-uint16 test-uint16-emulation
    test-casts'
    test-casts-uint'
in PolyML

```

```
end
```

```

theory Native-Word-Test-PolyML64 imports
    Native-Word-Test
begin

```

```

test-code test-uint64' = 0x12
in PolyML

```

```
<ML>
```

```
end
```

```

theory Native-Word-Test-Scala imports
    Native-Word-Test
begin

```

12.3 Test with Scala

In Scala, *uint* and *uint32* are both implemented as type `Int`. When they are used in the same generated program, we have to suppress the type class instances for one of them.

code-printing class-instance *uint32* :: *equal* \rightarrow (*Scala*) –

```

test-code
    test-uint64 test-uint64' = 0x12
    test-uint32 test-uint32' = 0x12
    test-uint16
    test-uint8 test-uint8' = 0x12
    test-uint
    test-casts test-casts' test-casts''
    test-casts-uint test-casts-uint' test-casts-uint''
in Scala

```

```
end
```

```

theory Native-Word-Test-MLton imports
    Native-Word-Test
begin

```


12.4 Test with MLton

test-code

```

test-uint64 test-uint64' = 0x12
test-uint32 test-uint32' = 0x12
test-uint8 test-uint8' = 0x12
test-uint
test-casts
test-casts''
test-casts-uint
test-casts-uint''

```

in *MLton*

MLton provides `Word16` and `Word64` structures. To test them in the `SML_word` target, we have to associate a driver with the combination.

<ML>

test-code

```

test-uint64 test-wint64' = 0x12
test-uint32 test-wint32' = 0x12
test-uint16
test-uint8 test-uint8' = 0x12
test-uint
test-casts
test-casts'
test-casts''
test-casts-uint
test-casts-uint'
test-casts-uint''

```

in *MLton-word*

end

theory *Native-Word-Test-MLton2* **imports**

Native-Word-Test-Emu

begin

export-code *test-casts'* in *SML* **module-name** *Generated-Code*

test-code

```

test-uint16 test-wint16-emulation
test-casts'
test-casts-uint'

```

in *MLton*

end

Chapter 13

User guide for native words

This tutorial explains how to best use the types for native words like *wint32* in your formalisation. You can base your formalisation

1. either directly on these types,
2. or on the generic *'a word* and only introduce native words a posteriori via code generator refinement.

The first option causes the least overhead if you have to prove only little about the words you use and start a fresh formalisation. Just use the native type *wint32* instead of *32 word* and similarly for *wint64*, *wint16*, and *wint8*. As native word types are meant only for code generation, the lemmas about *'a word* have not been duplicated, but you can transfer theorems between native word types and *'a word* using the transfer package.

Note, however, that this option restricts your work a bit: your own functions cannot be “polymorphic” in the word length, but you have to define a separate function for every word length you need.

The second option is recommended if you already have a formalisation based on *'a word* or if your proofs involve words and their properties. It separates code generation from modelling and proving, i.e., you can work with words as usual. Consequently, you have to manually setup the code generator to use the native types wherever you want. The following describes how to achieve this with moderate effort.

Note, however, that some target languages of the code generator (especially OCaml) do not support all the native word types provided. Therefore, you should only import those types that you need – the theory file for each type mentions at the top the restrictions for code generation. For example, PolyML does not provide the *Word16* structure, and OCaml provides neither *Word8* nor *Word16*. You can still use these theories provided that you also import the theory *Native-Word.Code-Target-Int-Bit* (which implements

int by target-language integers), but these words will be implemented via Isabelle’s *Word* library, i.e., you do not gain anything in terms of efficiency.

There is a separate code target *SML-word* for *SML*. If you use one of the native words that PolyML does not support (such as *uint16* and *uint64* in 32-bit mode), but would like to map its operations to the Standard Basis Library functions, make sure to use the target *SML-word* instead of *SML*; if you only use native word sizes that PolyML supports, you can stick with *SML*. This ensures that code generation within Isabelle as used by *Quickcheck*, *value* and `@{code}` in ML blocks continues to work.

13.1 Lifting functions from ‘a word to native words

This section shows how to convert functions from ‘a word to native words. For example, the following function *sum-squares* computes the sum of the first *n* square numbers in 16 bit arithmetic using a tail-recursive function *gen-sum-squares* with accumulator; for convenience, *sum-squares-int* takes an integer instead of a word.

function *gen-sum-squares* :: 16 word \Rightarrow 16 word \Rightarrow 16 word **where**
gen-sum-squares accum n =
*(if n = 0 then accum else gen-sum-squares (accum + n * n) (n - 1))*⟨proof⟩⟨proof⟩
definition *sum-squares* :: 16 word \Rightarrow 16 word **where**
sum-squares = gen-sum-squares 0

definition *sum-squares-int* :: int \Rightarrow 16 word **where**
sum-squares-int n = sum-squares (word-of-int n)

The generated code for *sum-squares* and *sum-squares-int* emulates words with unbounded integers and explicit modulus as specified in the theory *HOL-Library.Word*. But for efficiency, we want that the generated code uses machine words and machine arithmetic. Unfortunately, as ‘a word is polymorphic in the word length, the code generator can only do this if we use another type for machine words. The theory *Native-Word.Uint16* defines the type *uint16* for machine words of 16 bits. We just have to follow two steps to use it:

First, we lift all our functions from 16 word to *uint16*, i.e., *sum-squares*, *gen-sum-squares*, and *sum-squares-int* in our case. The theory *Native-Word.Uint16* sets up the lifting package for this and has already taken care of the arithmetic and bit-wise operations.

lift-definition *gen-sum-squares-uint* :: uint16 \Rightarrow uint16 \Rightarrow uint16
is *gen-sum-squares* ⟨proof⟩

lift-definition *sum-squares-uint* :: uint16 \Rightarrow uint16 **is** *sum-squares* ⟨proof⟩

lift-definition *sum-squares-int-uint* :: int \Rightarrow uint16 **is** *sum-squares-int* ⟨proof⟩

Second, we also have to transfer the code equations for our functions. The

attribute *Transfer.transferred* takes care of that, but it is better to check that the transfer succeeded: inspect the theorem to check that the new constants are used throughout.

```
lemmas [Transfer.transferred, code] =
  gen-sum-squares.simps
  sum-squares-def
  sum-squares-int-def
```

Finally, we export the code to standard ML. We use the target *SML-word* instead of *SML* to have the operations on *uint16* mapped to the Standard Basis Library. As PolyML does not provide a *Word16* type, the mapping for *uint16* is only active in the refined target *SML-word*.

```
export-code sum-squares-int-uint in SML-word
```

Nevertheless, we can still evaluate terms with *uint16* within Isabelle, i.e., PolyML, but this will be translated to *16 word* and therefore less efficient.

```
value sum-squares-int-uint 40
```

13.2 Storing native words in datatypes

The above lifting is necessary for all functions whose type mentions the word type. Fortunately, we do not have to duplicate functions that merely operate on datatypes that contain words. Nevertheless, we have to tell the code generator that these functions should call the new ones, which operate on machine words. This section shows how to achieve this with data refinement.

13.2.1 Example: expressions and two semantics

As the running example, we consider a language of expressions (literal values, less-than comparisons and conditional) where values are either booleans or 32-bit words. The original specification uses the type *32 word*.

```
datatype val = Bool bool | Word 32 word
datatype expr = Lit val | LT expr expr | IF expr expr expr
```

```
abbreviation (input) word :: 32 word ⇒ expr where word i ≡ Lit (Word i)
```

```
abbreviation (input) bool :: bool ⇒ expr where bool i ≡ Lit (Bool i)
```

— Denotational semantics of expressions, *None* denotes a type error

```
fun eval :: expr ⇒ val option where
  eval (Lit v) = Some v
| eval (LT e1 e2) =
  (case (eval e1, eval e2)
   of (Some (Word i1), Some (Word i2)) ⇒ Some (Bool (i1 < i2))
   | - ⇒ None)
| eval (IF e1 e2 e3) =
```

(*case eval* e_1 of *Some* (*Bool* b) \Rightarrow if b then *eval* e_2 else *eval* e_3
 | - \Rightarrow *None*)

— Small-step semantics of expressions, it gets stuck upon type errors.

inductive step :: *expr* \Rightarrow *expr* \Rightarrow *bool* (- \rightarrow - [50, 50] 60) **where**

$e \rightarrow e' \Longrightarrow$ *LT* e $e_2 \rightarrow$ *LT* $e' e_2$
 | $e \rightarrow e' \Longrightarrow$ *LT* (*word* i) $e \rightarrow$ *LT* (*word* i) e'
 | *LT* (*word* i_1) (*word* i_2) \rightarrow *bool* ($i_1 < i_2$)
 | $e \rightarrow e' \Longrightarrow$ *IF* e e_1 $e_2 \rightarrow$ *IF* $e' e_1 e_2$
 | *IF* (*bool* *True*) e_1 $e_2 \rightarrow e_1$
 | *IF* (*bool* *False*) e_1 $e_2 \rightarrow e_2$

— Compile the inductive definition with the predicate compiler

code-pred (*modes*: $i \Rightarrow o \Rightarrow$ *bool* as *reduce*, $i \Rightarrow i \Rightarrow$ *bool* as *step'*) *step* \langle *proof* \rangle

13.2.2 Change the datatype to use machine words

Now, we want to use *uint32* instead of *32 word*. The goal is to make the code generator use the new type without duplicating any of the types (*val*, *expr*) or the functions (*eval*, *reduce*) on such types.

The constructor *Word* has *32 word* in its type, so we have to lift it to *Word'*, and the same holds for the case combinator *case-val*, which *case-val'* replaces.¹ Next, we set up the code generator accordingly: *Bool* and *Word'* are the new constructors for *val*, and *case-val'* is the new case combinator with an appropriate case certificate.² We delete the code equations for the old constructor *Word* and case combinator *case-val* such that the code generator reports missing adaptations.

lift-definition *Word'* :: *uint32* \Rightarrow *val* **is** *Word* \langle *proof* \rangle

code-datatype *Bool* *Word'*

¹Note that we should not declare a case translation for the new case combinator because this will break parsing case expressions with old case combinator.

²Case certificates tell the code generator to replace the HOL case combinator for a datatype with the case combinator of the target language. Without a case certificate, the code generator generates a function that re-implements the case combinator; in a strict languages like ML or Scala, this means that the code evaluates all possible cases before it decides which one is taken.

Case certificates are described in Haftmann's PhD thesis [1, Def. 27]. For a datatype *dt* with constructors C_1 to C_n where each constructor C_i takes k_i parameters, the certificate for the case combinator *case-dt* looks as follows:

lemma

assumes *CASE* \equiv *dt-case* c_1 c_2 ... c_n
shows (*CASE* (C_1 a_{11} a_{12} ... a_{1k_1}) \equiv c_1 a_{11} a_{12} ... a_{1k_1})
 &&& (*CASE* (C_2 a_{21} a_{22} ... a_{2k_2}) \equiv c_2 a_{21} a_{22} ... a_{2k_2})
 &&& ...
 &&& (*CASE* (C_n a_{n1} a_{n2} ... a_{nk_n}) \equiv c_n a_{n1} a_{n2} ... a_{nk_n})

lift-definition *case-val'* :: (*bool* \Rightarrow 'a) \Rightarrow (*uint32* \Rightarrow 'a) \Rightarrow *val* \Rightarrow 'a **is** *case-val*
 <proof>

lemmas [*code*, *simp*] = *val.case* [*Transfer.transferred*]

lemma *case-val'-cert*:

fixes *bool word' b w*

assumes *CASE* \equiv *case-val' bool word'*

shows (*CASE* (*Bool b*) \equiv *bool b*) &&& (*CASE* (*Word' w*) \equiv *word' w*)

<proof>

<ML>

declare [[*code drop: case-val Word*]]

13.2.3 Make functions use functions on machine words

Finally, we merely have to change the code equations to use the new functions that operate on *uint32*. As before, the attribute *Transfer.transferred* does the job. In our example, we adapt the equality test on *val* (code equations *val.eq.simps*) and the denotational and small-step semantics (code equations *eval.simps* and *step.equation*, respectively).

We check that the adaptation has succeeded by exporting the functions. As we only use native word sizes that PolyML supports, we can use the usual target *SML* instead of *SML-word*.

lemmas [*code*] =

val.eq.simps[*THEN meta-eq-to-obj-eq, Transfer.transferred, THEN eq-reflection*]

eval.simps[*Transfer.transferred*]

step.equation[*Transfer.transferred*]

export-code *reduce step' eval checking SML*

13.3 Troubleshooting

This section explains some possible problems when using native words. If you experience other difficulties, please contact the author.

13.3.1 *export-code* raises an exception

Probably, you have defined and are using a function on a native word type, but the code equation refers to emulated words. For example, the following defines a function *double* that doubles a word. When we try to export code for *double* without any further setup, *export-code* will raise an exception or generate code that does not compile.

lift-definition *double* :: *uint32* \Rightarrow *uint32* **is** $\lambda x. x + x$ <proof>

We have to prove a code equation that only uses the existing operations on *uint32*. Then, *export-code* works again.

lemma *double-code* [*code*]: *double n = n + n*
<proof>

13.3.2 The generated code does not compile

Probably, you have been exporting to a target language for which there is no setup, or your compiler does not provide the required API. Every theory for native words mentions at the start the limitations on code generation. Check that your concrete application meets all the requirements.

Alternatively, this might be an instance of the problem described in §13.3.1. For Haskell, you have to enable the extension `TypeSynonymInstances` with `-XTypeSynonymInstances` if you are using polymorphic bit operations on the native word types.

13.3.3 The generated code is too slow

The generated code will most likely not be as fast as a direct implementation in the target language with manual tuning. This is because we want the configuration of the code generation to be sound (as it can be used to prove theorems in Isabelle). Therefore, the bit operations sometimes perform range checks before they call the target language API. Here are some examples:

- Shift distances and bit indices in target languages are often expected to fit into a bounded integer or word. However, the size of these types varies across target languages and platforms. Hence, no Isabelle/HOL type can model uniformly all of them. Instead, the bit operations use arbitrary-precision integers for such quantities and check at run-time that the values fit into a bounded integer or word, respectively – if not, they raise an exception.
- Division and modulo operations explicitly test whether the divisor is 0 and return the HOL value of division by 0 in that case. This is necessary because some languages leave the behaviour of division by 0 unspecified.

If you have better ideas how to eliminate such checks and speed up the generated code without sacrificing soundness, please contact the author!

Bibliography

- [1] F. Haftmann. *Code Generation from Specifications in Higher-Order-Logic*. PhD thesis, Institut für Informatik, Technische Universität München, 2009.