

# Native Words

Andreas Lochbihler

April 12, 2026

**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

<b>1</b>	<b>Common logic auxiliary for all fixed-width word types</b>	<b>5</b>
1.1	Some abstract nonsense . . . . .	5
1.2	Establishing type class instances for type copies of word type	5
1.3	Establishing operation variants tailored towards target languages . . . . .	10
1.3.1	Division by signed division . . . . .	12
1.3.2	Conversion from <i>int</i> to <i>'a word</i> . . . . .	13
1.3.3	Quickcheck conversion functions . . . . .	13
1.3.4	Code generator setup . . . . .	14
<b>2</b>	<b>Common base for target language implementations of word types</b>	<b>17</b>
2.1	SML . . . . .	17
2.2	Haskell . . . . .	17
<b>3</b>	<b>Unsigned words of 64 bits</b>	<b>21</b>
3.1	Type definition and primitive operations . . . . .	21
3.2	Code setup . . . . .	24
3.3	Quickcheck setup . . . . .	34
<b>4</b>	<b>Unsigned words of 32 bits</b>	<b>35</b>
4.1	Type definition and primitive operations . . . . .	35
4.2	Code setup . . . . .	37
4.3	Quickcheck setup . . . . .	46
<b>5</b>	<b>Unsigned words of 16 bits</b>	<b>47</b>
5.1	Type definition and primitive operations . . . . .	47
5.2	Code setup . . . . .	50
5.3	Quickcheck setup . . . . .	55
<b>6</b>	<b>Unsigned words of 8 bits</b>	<b>57</b>
6.1	Type definition and primitive operations . . . . .	57
6.2	Code setup . . . . .	59
6.3	Quickcheck setup . . . . .	66

<b>7</b>	<b>Unsigned words of default size</b>	<b>67</b>
7.1	Type definition and primitive operations . . . . .	68
7.2	Code setup . . . . .	70
7.3	Quickcheck setup . . . . .	80
<b>8</b>	<b>Conversions between unsigned words and between char</b>	<b>81</b>
8.1	Conversion between native words . . . . .	81
8.2	Compatibility with Imperative/HOL . . . . .	86
<b>9</b>	<b>Test cases</b>	<b>87</b>
9.1	Tests for	
	$\text{isa}^{\wedge}\text{typinteger}$ . . . . .	87
9.2	Tests for <i>uint8</i> . . . . .	88
9.3	Tests for <i>uint16</i> . . . . .	90
9.4	Tests for <i>uint32</i> . . . . .	92
9.5	Tests for <i>uint64</i> . . . . .	94
9.6	Tests for <i>uint</i> . . . . .	96
9.7	Tests for casts . . . . .	98
9.8	Test cases for emulation of native words . . . . .	100
	9.8.1 Tests for <i>uint8</i> . . . . .	100
	9.8.2 Tests for <i>uint16</i> . . . . .	100
9.9	Test with PolyML . . . . .	100
9.10	Test with Scala . . . . .	101
9.11	Test with GHC . . . . .	102
	9.11.1 Test quickcheck narrowing . . . . .	102
9.12	Test with MLton . . . . .	103
9.13	Test with OCaml . . . . .	104
9.14	Test with SML/NJ . . . . .	105
<b>10</b>	<b>User guide for native words</b>	<b>107</b>
10.1	Lifting functions from <i>'a word</i> to native words . . . . .	108
10.2	Storing native words in datatypes . . . . .	109
	10.2.1 Example: expressions and two semantics . . . . .	109
	10.2.2 Change the datatype to use machine words . . . . .	110
	10.2.3 Make functions use functions on machine words . . . . .	111
10.3	Troubleshooting . . . . .	111
	10.3.1 <i>export-code</i> raises an exception . . . . .	111
	10.3.2 The generated code does not compile . . . . .	112
	10.3.3 The generated code is too slow . . . . .	112

# Chapter 1

## Common logic auxiliary for all fixed-width word types

```
theory Uint-Common
imports
  HOL-Library.Word
  Word-Lib.Signed-Division-Word
  Word-Lib.Most-significant-bit
  Word-Lib.Bit-Comprehension
begin
```

### 1.1 Some abstract nonsense

```
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]
```

### 1.2 Establishing type class instances for type copies of word type

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 ::  $\langle 'a \Rightarrow 'b \text{ word} \rangle$ 
```

## 6 CHAPTER 1. COMMON LOGIC AUXILIARY FOR ALL FIXED-WIDTH WORD TYPES

```

assumes type-definition: ⟨type-definition word-of of-word UNIV⟩
begin

lemma word-of-word:
  ⟨word-of (of-word w) = w⟩
  ⟨proof⟩

lemma of-word-of [code abstype]:
  ⟨of-word (word-of p) = p⟩
  — Use an abstract type for code generation to disable pattern matching on
    of-word.
  ⟨proof⟩

lemma word-of-eqI:
  ⟨p = q⟩ if ⟨word-of p = word-of q⟩
  ⟨proof⟩

lemma eq-iff-word-of:
  ⟨p = q ↔ word-of p = word-of q⟩
  ⟨proof⟩

lemma inj-word-of:
  ⟨inj word-of⟩
  ⟨proof⟩

end

bundle constraintless
begin

  ⟨ML⟩

end

locale word-type-copy-ring = word-type-copy
opening constraintless +
constrains word-of :: ⟨'a ⇒ 'b::len word⟩
assumes word-of-0 [code]: ⟨word-of 0 = 0⟩
and word-of-1 [code]: ⟨word-of 1 = 1⟩
and word-of-add [code]: ⟨word-of (p + q) = word-of p + word-of q⟩
and word-of-minus [code]: ⟨word-of (- p) = - (word-of p)⟩
and word-of-diff [code]: ⟨word-of (p - q) = word-of p - word-of q⟩
and word-of-mult [code]: ⟨word-of (p * q) = word-of p * word-of q⟩
and word-of-div [code]: ⟨word-of (p div q) = word-of p div word-of q⟩
and word-of-mod [code]: ⟨word-of (p mod q) = word-of p mod word-of q⟩
and equal-iff-word-of [code]: ⟨HOL.equal p q ↔ HOL.equal (word-of p)
(word-of q)⟩
and less-eq-iff-word-of [code]: ⟨p ≤ q ↔ word-of p ≤ word-of q⟩
and less-iff-word-of [code]: ⟨p < q ↔ word-of p < word-of q⟩

```

## 1.2. ESTABLISHING TYPE CLASS INSTANCES FOR TYPE COPIES OF WORD TYPE7

```
and word-of-bot [code]: ⟨word-of bot = bot⟩
and word-of-top [code]: ⟨word-of top = top⟩
begin

lemma of-class-comm-ring-1:
  ⟨OFCLASS('a, comm-ring-1-class)⟩
  ⟨proof⟩

lemma of-class-comm-semiring-1-cancel:
  ⟨OFCLASS('a, comm-semiring-1-cancel-class)⟩
  ⟨proof⟩

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

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

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

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

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

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

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

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

end

locale word-type-copy-bits = word-type-copy-ring
  opening constraintless and bit-operations-syntax +
  constrains word-of :: ⟨a::{comm-ring-1, semiring-modulo, equal, linorder} ⇒
```

## 8CHAPTER 1. COMMON LOGIC AUXILIARY FOR ALL FIXED-WIDTH WORD TYPES

```

'b::len word>
fixes signed-drop-bit :: ⟨nat ⇒ 'a ⇒ 'a⟩
assumes bit-eq-word-of: ⟨bit p = bit (word-of p)⟩
and word-of-not [code]: ⟨word-of (NOT p) = NOT (word-of p)⟩
and word-of-and [code]: ⟨word-of (p AND q) = word-of p AND word-of q⟩
and word-of-or [code]: ⟨word-of (p OR q) = word-of p OR word-of q⟩
and word-of-xor [code]: ⟨word-of (p XOR q) = word-of p XOR word-of q⟩
and word-of-mask [code]: ⟨word-of (mask n) = mask n⟩
and word-of-push-bit: ⟨word-of (push-bit n p) = push-bit n (word-of p)⟩
and word-of-drop-bit: ⟨word-of (drop-bit n p) = drop-bit n (word-of p)⟩
and word-of-signed-drop-bit: ⟨word-of (signed-drop-bit n p) = Word.signed-drop-bit
n (word-of p)⟩
and word-of-take-bit [code]: ⟨word-of (take-bit n p) = take-bit n (word-of p)⟩
and word-of-set-bit [code]: ⟨word-of (Bit-Operations.set-bit n p) = Bit-Operations.set-bit
n (word-of p)⟩
and word-of-unset-bit [code]: ⟨word-of (unset-bit n p) = unset-bit n (word-of
p)⟩
and word-of-flip-bit [code]: ⟨word-of (flip-bit n p) = flip-bit n (word-of p)⟩
begin

lemma word-of-bool:
  ⟨word-of (of-bool n) = of-bool n⟩
  ⟨proof⟩

lemma word-of-nat:
  ⟨word-of (of-nat n) = of-nat n⟩
  ⟨proof⟩

lemma word-of-numeral [simp]:
  ⟨word-of (numeral n) = numeral n⟩
  ⟨proof⟩

lemma word-of-power:
  ⟨word-of (p ^ n) = word-of p ^ n⟩
  ⟨proof⟩

lemma even-iff-word-of:
  ⟨2 dvd p ⟷ 2 dvd word-of p⟩ (is ⟨?P ⟷ ?Q⟩)
  ⟨proof⟩

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

lemma [code]:
  ⟨take-bit n a = a AND mask n⟩ for a :: 'a
  ⟨proof⟩

lemma [code]:

```

## 1.2. ESTABLISHING TYPE CLASS INSTANCES FOR TYPE COPIES OF WORD TYPE9

```

⟨mask 0 = (0 :: 'a)⟩
⟨mask (Suc n) = push-bit n (1 :: 'a) OR mask n⟩
⟨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-int (int-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 and 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: <msb p <=> msb (word-of p)>
    and size-eq-word-of: <Nat.size (p :: 'a) = Nat.size (word-of p)>
    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-bit-comprehension:
  <OFCLASS('a, bit-comprehension-class)>
  <proof>

end

```

### 1.3 Establishing operation variants tailored towards target languages

```

locale word-type-copy-target-language = word-type-copy-misc +
  constrains word-of :: <'a::{ring-bit-operations, equal, linorder} => 'b::len word>
  fixes size-integer :: integer
    and almost-size :: nat
  assumes size-integer-eq-length: <size-integer = Nat.of-nat LENGTH('b::len)>
    and almost-size-eq-decr-length: <almost-size = LENGTH('b::len) - Suc 0>
begin

definition shiftl :: <'a => integer => 'a>

```

### 1.3. ESTABLISHING OPERATION VARIANTS TAILORED TOWARDS TARGET LANGUAGES11

**where**  $\langle \text{shiffl } w \ k = (\text{if } k < 0 \vee \text{size-integer} \leq k \text{ then undefined } (\text{push-bit} :: \text{nat} \Rightarrow 'a \Rightarrow 'a) \ w \ k$   
 $\Rightarrow 'a \Rightarrow 'a) \ w \ k$   
 $\text{else push-bit } (\text{nat-of-integer } k) \ w) \rangle$

**lemma** *word-of-shiffl* [code abstract]:

$\langle \text{word-of } (\text{shiffl } w \ k) =$   
 $(\text{if } k < 0 \vee \text{size-integer} \leq k \text{ then word-of } (\text{undefined } (\text{push-bit} :: - \Rightarrow - \Rightarrow 'a) \ w$   
 $k)$   
 $\text{else push-bit } (\text{nat-of-integer } k) \ (\text{word-of } w)) \rangle$   
 $\langle \text{proof} \rangle$

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

$\langle \text{push-bit } k \ w = (\text{if } k < \text{size} \text{ then shiffl } w \ (\text{integer-of-nat } k) \ \text{else } 0) \rangle$   
 $\langle \text{proof} \rangle$

**definition** *shiftr* ::  $\langle 'a \Rightarrow \text{integer} \Rightarrow 'a \rangle$

**where**  $\langle \text{shiftr } w \ k = (\text{if } k < 0 \vee \text{size-integer} \leq k \text{ then undefined } (\text{drop-bit} :: \text{nat} \Rightarrow 'a \Rightarrow 'a) \ w \ k$   
 $\Rightarrow 'a \Rightarrow 'a) \ w \ k$   
 $\text{else drop-bit } (\text{nat-of-integer } k) \ w) \rangle$

**lemma** *word-of-shiftr* [code abstract]:

$\langle \text{word-of } (\text{shiftr } w \ k) =$   
 $(\text{if } k < 0 \vee \text{size-integer} \leq k \text{ then word-of } (\text{undefined } (\text{drop-bit} :: - \Rightarrow - \Rightarrow 'a) \ w$   
 $k)$   
 $\text{else drop-bit } (\text{nat-of-integer } k) \ (\text{word-of } w)) \rangle$   
 $\langle \text{proof} \rangle$

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

$\langle \text{drop-bit } k \ w = (\text{if } k < \text{size} \text{ then shiftr } w \ (\text{integer-of-nat } k) \ \text{else } 0) \rangle$   
 $\langle \text{proof} \rangle$

**definition** *sshiftr* ::  $\langle 'a \Rightarrow \text{integer} \Rightarrow 'a \rangle$

**where**  $\langle \text{sshiftr } w \ k = (\text{if } k < 0 \vee \text{size-integer} \leq k \text{ then undefined } (\text{signed-drop-bit} :: - \Rightarrow - \Rightarrow 'a) \ w \ k$   
 $:: - \Rightarrow - \Rightarrow 'a) \ w \ k$   
 $\text{else signed-drop-bit } (\text{nat-of-integer } k) \ w) \rangle$

**lemma** *word-of-sshiftr* [code abstract]:

$\langle \text{word-of } (\text{sshiftr } w \ k) =$   
 $(\text{if } k < 0 \vee \text{size-integer} \leq k \text{ then word-of } (\text{undefined } (\text{signed-drop-bit} :: - \Rightarrow - \Rightarrow 'a) \ w \ k)$   
 $'a) \ w \ k)$   
 $\text{else Word.signed-drop-bit } (\text{nat-of-integer } k) \ (\text{word-of } w)) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *signed-drop-bit-code* [code]:

$\langle \text{signed-drop-bit } k \ w = (\text{if } k < \text{size} \text{ then sshiftr } w \ (\text{integer-of-nat } k)$   
 $\text{else if } (\text{bit } w \ \text{almost-size}) \ \text{then } - \ 1 \ \text{else } 0) \rangle$   
 $\langle \text{proof} \rangle$

**definition** *test-bit* ::  $\langle 'a \Rightarrow \text{integer} \Rightarrow \text{bool} \rangle$

**where**  $\langle \text{test-bit } w \ k = (\text{if } k < 0 \vee \text{size-integer} \leq k \text{ then undefined } (\text{bit} :: 'a \Rightarrow -) \ w \ k$   
 $\text{else bit } w \ (\text{nat-of-integer } k)) \rangle$

**lemma** *test-bit-eq* [code]:  
 $\langle \text{test-bit } w \ k = (\text{if } k < 0 \vee \text{size-integer} \leq k \text{ then undefined } (\text{bit} :: 'a \Rightarrow -) \ w \ k$   
 $\text{else bit } (\text{word-of } w) \ (\text{nat-of-integer } k)) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *bit-code* [code]:  
 $\langle \text{bit } w \ n \longleftrightarrow n < \text{size} \wedge \text{test-bit } w \ (\text{integer-of-nat } n) \rangle$   
 $\langle \text{proof} \rangle$

**end**

### 1.3.1 Division by signed division

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

**context**  
**begin**

**private lemma** *div-half-nat*:  
**fixes**  $m \ n :: \text{nat}$   
**assumes**  $n \neq 0$   
**shows**  $(m \ \text{div} \ n, \ m \ \text{mod} \ n) =$   
 $\text{let}$   
 $q = 2 * (\text{drop-bit } 1 \ m \ \text{div} \ n);$   
 $r = m - q * n$   
 $\text{in if } n \leq r \text{ then } (q + 1, r - n) \text{ else } (q, r)$   
 $\langle \text{proof} \rangle$  **lemma** *div-half-word*:  
**fixes**  $x \ y :: 'a :: \text{len word}$   
**assumes**  $y \neq 0$   
**shows**  $(x \ \text{div} \ y, \ x \ \text{mod} \ y) =$   
 $\text{let}$   
 $q = \text{push-bit } 1 \ (\text{drop-bit } 1 \ x \ \text{div} \ y);$   
 $r = x - q * y$   
 $\text{in if } y \leq r \text{ then } (q + 1, r - y) \text{ else } (q, r)$   
 $\langle \text{proof} \rangle$

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

```

else let
  q = (push-bit 1 (drop-bit 1 x sdiv y));
  r = x - q * y
  in if y ≤ r then (q + 1, r - y) else (q, r)
⟨proof⟩

end

```

### 1.3.2 Conversion from *int* to *'a* word

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

**includes** *bit-operations-syntax*

**assumes** *shift-def*: *shift* = *push-bit* *LENGTH('a)* 1

**and** *overflow-def*: *overflow* = *push-bit* (*LENGTH('a)* - 1) 1

**shows**

```

(word-of-int i :: 'a :: len word) =
  (let i' = i AND mask LENGTH('a)
   in if bit i' (LENGTH('a) - 1) then
     if i' - shift < - overflow ∨ overflow ≤ i' - shift then arbitrary1 i' else
word-of-int (i' - shift)
     else if i' < - overflow ∨ overflow ≤ i' then arbitrary2 i' else word-of-int i')
⟨proof⟩

```

### 1.3.3 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) ◦→ ( $\lambda k. \text{Pair}$  (

let *n* = *a-of-natural* *k*

in (*n*,  $\lambda-. \text{Code-Evaluation.term-of } n$ ))

**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

```

### 1.3.4 Code generator setup

```
code-identifier code-module Uint-Common →
```

### 1.3. ESTABLISHING OPERATION VARIANTS TAILORED TOWARDS TARGET LANGUAGES<sup>15</sup>

*(SML) Word* **and** *(Haskell) Word* **and** *(OCaml) Word* **and** *(Scala) Word*

**end**



## Chapter 2

# Common base for target language implementations of word types

```
theory Code-Target-Word
  imports HOL-Library.Word
begin
```

### 2.1 SML

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

*<ML>*

### 2.2 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.

```
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)*;

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

*shifflBounded* :: *Data.Bits.Bits a => a -> Integer -> a*;  
*shifflBounded* *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;
```

```
(.|.) :: 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;
```

```
shiftlUnbounded :: Data.Bits.Bits a => a -> Prelude.Int -> a;
```

```
shiftlUnbounded = Data.Bits.shiftL;
```

```
shiftlBounded :: Data.Bits.Bits a => a -> Prelude.Int -> a;
```

```
shiftlBounded = 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*

**end**

## Chapter 3

# Unsigned words of 64 bits

```
theory Uint64
imports
  Uint-Common
  Code-Target-Word
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.

### 3.1 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 ::  $\langle \{ comm\text{-}ring\text{-}1, semiring\text{-}modulo, equal, linorder, order\text{-}bot, order\text{-}top \} \rangle$ 
begin
```

**lift-definition** *zero-uint64* :: *uint64* is 0 *<proof>*  
**lift-definition** *one-uint64* :: *uint64* is 1 *<proof>*  
**lift-definition** *plus-uint64* :: *<uint64 ⇒ uint64 ⇒ uint64>* is *<(+)>* *<proof>*  
**lift-definition** *uminus-uint64* :: *<uint64 ⇒ uint64>* is *uminus* *<proof>*  
**lift-definition** *minus-uint64* :: *<uint64 ⇒ uint64 ⇒ uint64>* is *<(-)>* *<proof>*  
**lift-definition** *times-uint64* :: *<uint64 ⇒ uint64 ⇒ uint64>* is *<(\*)>* *<proof>*  
**lift-definition** *divide-uint64* :: *<uint64 ⇒ uint64 ⇒ uint64>* is *<(div)>* *<proof>*  
**lift-definition** *modulo-uint64* :: *<uint64 ⇒ uint64 ⇒ uint64>* is *<(mod)>* *<proof>*  
**lift-definition** *equal-uint64* :: *<uint64 ⇒ uint64 ⇒ bool>* is *<HOL.equal>* *<proof>*  
**lift-definition** *less-eq-uint64* :: *<uint64 ⇒ uint64 ⇒ bool>* is *<(≤)>* *<proof>*  
**lift-definition** *less-uint64* :: *<uint64 ⇒ uint64 ⇒ bool>* is *<(<)>* *<proof>*  
**lift-definition** *bot-uint64* :: *uint64* is *bot* *<proof>*  
**lift-definition** *top-uint64* :: *uint64* is *top* *<proof>*

**global-interpretation** *uint64*: *word-type-copy-ring Abs-uint64 Rep-uint64*  
*<proof>*

**instance** *<proof>*

**end**

**instance** *uint64* :: *<{interval-bot, interval-top}>*  
*<proof>*

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

**lift-definition** *bit-uint64* :: *<uint64 ⇒ nat ⇒ bool>* is *bit* *<proof>*  
**lift-definition** *not-uint64* :: *<uint64 ⇒ uint64>* is *<Bit-Operations.not>* *<proof>*  
**lift-definition** *and-uint64* :: *<uint64 ⇒ uint64 ⇒ uint64>* is *<Bit-Operations.and>*  
*<proof>*  
**lift-definition** *or-uint64* :: *<uint64 ⇒ uint64 ⇒ uint64>* is *<Bit-Operations.or>*  
*<proof>*  
**lift-definition** *xor-uint64* :: *<uint64 ⇒ uint64 ⇒ uint64>* is *<Bit-Operations.xor>*  
*<proof>*  
**lift-definition** *mask-uint64* :: *<nat ⇒ uint64>* is *mask* *<proof>*  
**lift-definition** *push-bit-uint64* :: *<nat ⇒ uint64 ⇒ uint64>* is *push-bit* *<proof>*  
**lift-definition** *drop-bit-uint64* :: *<nat ⇒ uint64 ⇒ uint64>* is *drop-bit* *<proof>*  
**lift-definition** *signed-drop-bit-uint64* :: *<nat ⇒ uint64 ⇒ uint64>* is *signed-drop-bit*  
*<proof>*  
**lift-definition** *take-bit-uint64* :: *<nat ⇒ uint64 ⇒ uint64>* is *take-bit* *<proof>*  
**lift-definition** *set-bit-uint64* :: *<nat ⇒ uint64 ⇒ uint64>* is *Bit-Operations.set-bit*  
*<proof>*  
**lift-definition** *unset-bit-uint64* :: *<nat ⇒ uint64 ⇒ uint64>* is *unset-bit* *<proof>*  
**lift-definition** *flip-bit-uint64* :: *<nat ⇒ uint64 ⇒ uint64>* is *flip-bit* *<proof>*

**global-interpretation** *uint64*: *word-type-copy-bits Abs-uint64 Rep-uint64 signed-drop-bit-uint64*  
*<proof>*

**instance**

⟨*proof*⟩

**end**

**lift-definition** *uint64-of-nat* :: ⟨*nat* ⇒ *uint64*⟩

**is** *word-of-nat* ⟨*proof*⟩

**lift-definition** *nat-of-uint64* :: ⟨*uint64* ⇒ *nat*⟩

**is** *unat* ⟨*proof*⟩

**lift-definition** *uint64-of-int* :: ⟨*int* ⇒ *uint64*⟩

**is** *word-of-int* ⟨*proof*⟩

**lift-definition** *int-of-uint64* :: ⟨*uint64* ⇒ *int*⟩

**is** *uint* ⟨*proof*⟩

**context**

**includes** *integer.lifting*

**begin**

**lift-definition** *Uint64* :: ⟨*integer* ⇒ *uint64*⟩

**is** *word-of-int* ⟨*proof*⟩

**lift-definition** *integer-of-uint64* :: ⟨*uint64* ⇒ *integer*⟩

**is** *uint* ⟨*proof*⟩

**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*

⟨*proof*⟩

**instantiation** *uint64* :: {*size, msb, bit-comprehension*}

**begin**

**lift-definition** *size-uint64* :: ⟨*uint64* ⇒ *nat*⟩ **is** *size* ⟨*proof*⟩

**lift-definition** *msb-uint64* :: ⟨*uint64* ⇒ *bool*⟩ **is** *msb* ⟨*proof*⟩

**lift-definition** *set-bits-uint64* :: ⟨(*nat* ⇒ *bool*) ⇒ *uint64*⟩ **is** *set-bits* ⟨*proof*⟩

**lift-definition** *set-bits-aux-uint64* :: ⟨(*nat* ⇒ *bool*) ⇒ *nat* ⇒ *uint64* ⇒ *uint64*⟩ **is**

*set-bits-aux* ⟨*proof*⟩

**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*

⟨*proof*⟩

```
instance <proof>
```

```
end
```

### 3.2 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)  $\langle (*$  Test that words can handle numbers between 0 and 63  $*)$

```
val - = if 64 <= 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 shiftl : uint64 -> IntInf.int -> uint64;
  val shiftr : uint64 -> IntInf.int -> uint64;
  val shiftr-signed : uint64 -> IntInf.int -> 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;

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 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 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 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 ⇒ *uint64*

**where** *Uint64-signed* *i* = (if *i* < -(0x8000000000000000) ∨ *i* ≥ 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'* - 0x1000000000000000) else *Uint64-signed*  
*i'*)  
**including** *undefined-transfer* and *integer.lifting* ⟨proof⟩

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

*Rep-uint64* (*Uint64-signed* *i*) =  
 (if *i* < -(0x8000000000000000) ∨ *i* ≥ 0x8000000000000000 then *Rep-uint64*  
 (*undefined* *Uint64* *i*) else *word-of-int* (*int-of-integer* *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* (=) (λ*x*. *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** λ*x* :: *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.Type* **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 (STR "Numeral-Type.num1'") []]]], TR (STR "Uint64.uint64'")
[]))
  (term-of-class.term-of (Rep-uint64' x))
⟨proof⟩
```

**code-printing**

```
type-constructor uint64 →
  (SML) Uint64.uint64 and
  (Haskell) Uint64.Word64 and
  (OCaml) int64 and
  (Scala) Long
| constant Uint64 →
  (SML) Uint64.fromInt and
  (Haskell) (Prelude.fromInteger - :: Uint64.Word64) and
  (Haskell-Quickcheck) (Prelude.fromInteger (Prelude.toInteger -) :: Uint64.Word64)
and
  (Scala) -.longValue
| constant Uint64-signed →
  (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]:  $x \text{ div } y = (\text{if } y = 0 \text{ then } 0 \text{ else } \text{uint64-div } x \ y)$   
**including** *undefined-transfer* ⟨proof⟩

**lemma** *mod-uint64-code* [code]:  $x \text{ mod } y = (\text{if } y = 0 \text{ then } x \text{ else } \text{uint64-mod } x \ y)$   
**including** *undefined-transfer* ⟨proof⟩

**definition** *uint64-sdiv* ::  $\text{uint64} \Rightarrow \text{uint64} \Rightarrow \text{uint64}$   
**where**

*uint64-sdiv*  $x \ y =$   
 (*if*  $y = 0$  *then* *undefined*  $((\text{div}) :: \text{uint64} \Rightarrow -) \ x \ (0 :: \text{uint64})$   
*else* *Abs-uint64*  $(\text{Rep-uint64 } x \ \text{sdiv } \text{Rep-uint64 } y)$ )

**definition** *div0-uint64* ::  $\text{uint64} \Rightarrow \text{uint64}$

**where** [code abort]: *div0-uint64*  $x = \text{undefined } ((\text{div}) :: \text{uint64} \Rightarrow -) \ x \ (0 :: \text{uint64})$

**definition** *mod0-uint64* ::  $\text{uint64} \Rightarrow \text{uint64}$

**where** [code abort]: *mod0-uint64*  $x = \text{undefined } ((\text{mod}) :: \text{uint64} \Rightarrow -) \ x \ (0 :: \text{uint64})$

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

*uint64-divmod*  $x \ y =$   
 (*if*  $0x8000000000000000 \leq y$  *then* *if*  $x < y$  *then*  $(0, x)$  *else*  $(1, x - y)$   
*else if*  $y = 0$  *then*  $(\text{div0-uint64 } x, \text{mod0-uint64 } x)$   
*else let*  $q = \text{push-bit } 1 \ (\text{uint64-sdiv } (\text{drop-bit } 1 \ x) \ y);$   
      $r = x - q * y$   
   *in if*  $r \geq y$  *then*  $(q + 1, r - y)$  *else*  $(q, r)$ )

**including** *undefined-transfer* ⟨proof⟩

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

*Rep-uint64*  $(\text{uint64-sdiv } x \ y) =$   
 (*if*  $y = 0$  *then* *Rep-uint64*  $(\text{undefined } ((\text{div}) :: \text{uint64} \Rightarrow -) \ x \ (0 :: \text{uint64}))$   
*else* *Rep-uint64*  $x \ \text{sdiv } \text{Rep-uint64 } y$ )

⟨proof⟩

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

**code-printing**

**constant** *uint64-div*  $\rightarrow$   
 (SML) *Uint64.divide* **and**  
 (Haskell) *Prelude.div*  
| **constant** *uint64-mod*  $\rightarrow$   
 (SML) *Uint64.modulus* **and**  
 (Haskell) *Prelude.mod*  
| **constant** *uint64-divmod*  $\rightarrow$

```

(Haskell) divmod
| constant uint64-sdiv  $\rightarrow$ 
  (OCaml) Int64.div and
  (Scala) - '/' -

```

**global-interpretation** *uint64*: *word-type-copy-target-language 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 64 63

```

```

defines uint64-test-bit = uint64.test-bit
and uint64-shiffl = uint64.shiffl
and uint64-shiftr = uint64.shiftr
and uint64-sshiftr = uint64.sshiftr
⟨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)

```

**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 w => fn i => if i < 0 orelse i >= 64 then raise (Fail argument to
uint64'-shiffl out of bounds) else Uint64.shiffl w i)

```

**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 w => fn i => if i < 0 orelse i >= 64 then raise (Fail argument to
uint64'-shiftr out of bounds) else Uint64.shiftr w i)

```

**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 w => fn i => if i < 0 orelse i >= 64 then raise (Fail argument to
uint64'-shiftr'-signed out of bounds) else Uint64.shiftr'-signed w 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]:
   $integer-of-uint64\ n =$ 
   $(if\ bit\ n\ 63\ then\ integer-of-uint64-signed\ (n\ AND\ 0x7FFFFFFFFFFFFFFF)\ OR\ 0x8000000000000000\ else\ integer-of-uint64-signed\ n)$ 
  <proof>
  including integer.lifting <proof>

end

code-printing
  constant integer-of-uint64  $\mapsto$ 
   $(SML)\ Uint64.toInt$  and

```

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

### 3.3 Quickcheck setup

**definition** *uint64-of-natural* :: *natural*  $\Rightarrow$  *uint64*  
**where** *uint64-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**  $\langle$ *proof* $\rangle$

**end**

**instantiation** *uint64* :: *narrowing* **begin**

**interpretation** *quickcheck-narrowing-samples*

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

*Typerep.Typerep* (*STR "Uint64.uint64"*) []  $\langle$ *proof* $\rangle$

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

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

**instance**  $\langle$ *proof* $\rangle$

**end**

**end**

# Chapter 4

## Unsigned words of 32 bits

```
theory Uint32
  imports
    Uint-Common
    Code-Target-Word
begin
```

### 4.1 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, order-bot,
  order-top}⟩
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⟩
```

**lift-definition** *bot-uint32* :: *uint32* **is** *bot* *<proof>*

**lift-definition** *top-uint32* :: *uint32* **is** *top* *<proof>*

**global-interpretation** *uint32*: *word-type-copy-ring Abs-uint32 Rep-uint32*  
*<proof>*

**instance** *<proof>*

**end**

**instance** *uint32* :: *<{interval-bot, interval-top}>*  
*<proof>*

**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 :: ⟨integer ⇒ uint32⟩
  is word-of-int ⟨proof⟩

lift-definition integer-of-uint32 :: ⟨uint32 ⇒ integer⟩
  is uint ⟨proof⟩

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
  ⟨proof⟩

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

lift-definition size-uint32 :: ⟨uint32 ⇒ nat⟩ is size ⟨proof⟩

lift-definition msb-uint32 :: ⟨uint32 ⇒ bool⟩ is msb ⟨proof⟩

lift-definition set-bits-uint32 :: ⟨(nat ⇒ bool) ⇒ uint32⟩ is set-bits ⟨proof⟩
lift-definition set-bits-aux-uint32 :: ⟨(nat ⇒ bool) ⇒ nat ⇒ uint32 ⇒ uint32⟩ is
set-bits-aux ⟨proof⟩

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
  ⟨proof⟩

instance ⟨proof⟩

end

```

## 4.2 Code setup

```

code-printing code-module Uint32 ↪ (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 shiftl : Word32.word -> IntInf.int -> Word32.word
    val shiftr : Word32.word -> IntInf.int -> Word32.word
    val shiftr-signed : Word32.word -> IntInf.int -> Word32.word
    val test-bit : Word32.word -> IntInf.int -> bool
end = struct

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 -> (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 -> (OCaml)

```

```

<module Uint32 : sig
    val less : int32 -> int32 -> bool
    val less-eq : int32 -> int32 -> bool
    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 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 shiftl(x: Int, n: BigInt) : Int = x << n.intValue

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

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

def test-bit(x: Int, n: BigInt) : Boolean =
  (x & (1 << n.intValue)) != 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  $\Rightarrow$  uint32  
**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* and *integer.lifting*  $\langle$ proof $\rangle$

**lemma** *Uint32-signed-code* [*code*]:  
*Rep-uint32* (*Uint32-signed*  $i$ ) =  
 (if  $i < -(0x80000000) \vee i \geq 0x80000000$  then *Rep-uint32* (undefined *Uint32*  $i$ )  
 else *word-of-int* (*int-of-integer*  $i$ ))  
 $\langle$ 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* (=) ( $\lambda x. x$ ) *Rep-uint32'*  
 $\langle$ proof $\rangle$

**lemma** *Rep-uint32'-code* [*code*]: *Rep-uint32'*  $x =$  (*BITS*  $n$ . bit  $x n$ )  
 $\langle$ proof $\rangle$

**lift-definition** *Abs-uint32'* :: 32 word  $\Rightarrow$  uint32 **is**  $\lambda x :: 32$  word.  $x$   $\langle$ proof $\rangle$

**lemma** *Abs-uint32'-code* [*code*]:  
*Abs-uint32'*  $x =$  *Uint32* (*integer-of-int* (*uint*  $x$ ))  
**including** *integer.lifting*  $\langle$ proof $\rangle$

**declare** [[*code drop*: *term-of-class.term-of* :: *uint32*  $\Rightarrow$  -]]

**lemma** *term-of-uint32-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 "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  $\rightarrow$ 
  (SML) Word32.word and
  (Haskell) Uint32.Word32 and
  (OCaml) int32 and
  (Scala) Int and
  (Eval) Word32.word
| constant Uint32  $\rightarrow$ 
  (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  $\rightarrow$ 
  (OCaml) Z.to'-int32
| constant 0 :: uint32  $\rightarrow$ 
  (SML) (Word32.fromInt 0) and
  (Haskell) (0 :: Uint32.Word32) and
  (OCaml) Int32.zero and
  (Scala) 0
| constant 1 :: uint32  $\rightarrow$ 
  (SML) (Word32.fromInt 1) and
  (Haskell) (1 :: Uint32.Word32) and
  (OCaml) Int32.one and
  (Scala) 1
| constant plus :: uint32  $\Rightarrow$  -  $\rightarrow$ 
  (SML) Word32.+ ((-), (-)) and
  (Haskell) infixl 6 + and
  (OCaml) Int32.add and
  (Scala) infixl 7 +
| constant uminus :: uint32  $\Rightarrow$  -  $\rightarrow$ 
  (SML) Word32.~ and
  (Haskell) negate and
  (OCaml) Int32.neg and
  (Scala) !(- -)
| constant minus :: uint32  $\Rightarrow$  -  $\rightarrow$ 
  (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*  $\Rightarrow$  *uint32*  $\Rightarrow$  *uint32*  
**where** *uint32-div* *x y* = *fst* (*uint32-divmod* *x y*)

**definition** *uint32-mod* :: *uint32*  $\Rightarrow$  *uint32*  $\Rightarrow$  *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*  $\langle$ *proof* $\rangle$

**lemma** *mod-uint32-code* [*code*]: *x mod y* = (*if y = 0 then x else uint32-mod x y*)  
**including** *undefined-transfer*  $\langle$ *proof* $\rangle$

**definition** *uint32-sdiv* :: *uint32*  $\Rightarrow$  *uint32*  $\Rightarrow$  *uint32*  
**where**

*uint32-sdiv* *x y* =  
(*if y = 0 then undefined ((div) :: uint32  $\Rightarrow$  -) x (0 :: uint32)*  
*else Abs-uint32 (Rep-uint32 x sdiv Rep-uint32 y)*)

**definition** *div0-uint32* :: *uint32*  $\Rightarrow$  *uint32*  
**where** [*code abort*]: *div0-uint32* *x* = *undefined ((div) :: uint32  $\Rightarrow$  -) x (0 :: uint32)*

**definition** *mod0-uint32* :: *uint32*  $\Rightarrow$  *uint32*  
**where** [*code abort*]: *mod0-uint32* *x* = *undefined ((mod) :: uint32  $\Rightarrow$  -) x (0 :: uint32)*

**lemma** *uint32-divmod-code* [*code*]:  
*uint32-divmod* *x y* =  
(*if 0x80000000  $\leq$  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  $\geq$  y then (q + 1, r - y) else (q, r)*)  
**including** *undefined-transfer*  $\langle$ *proof* $\rangle$

**lemma** *uint32-sdiv-code* [*code*]:  
*Rep-uint32 (uint32-sdiv x y)* =  
(*if y = 0 then Rep-uint32 (undefined ((div) :: uint32  $\Rightarrow$  -) x (0 :: uint32))*  
*else Rep-uint32 x sdiv Rep-uint32 y*)  
 $\langle$ *proof* $\rangle$

Note that we only need a translation for signed division, but not for the remainder because *uint32-divmod* *?x ?y* = (*if 2147483648  $\leq$  ?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  $\leq$  r then (q + 1, r - ?y) else (q, r)*) computes both with division only.

**code-printing**

**constant** *uint32-div*  $\rightarrow$

```

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

```

**global-interpretation** *uint32*: *word-type-copy-target-language 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 32 31

```

```

defines uint32-test-bit = uint32.test-bit
and uint32-shiffl = uint32.shiffl
and uint32-shiftr = uint32.shiftr
and uint32-sshiftr = uint32.sshiftr
⟨proof⟩

```

**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 => fn n => if n < 0 orelse 32 <= n then raise (Fail argument to
uint32'-test'-bit out of bounds) else Uint32.test'-bit w n)

```

**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 w => fn i => if i < 0 orelse i >= 32 then raise Fail argument to
uint32'-shiffl out of bounds else Uint32.shiffl w i)

```

**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 w => fn i => if i < 0 orelse i >= 32 then raise Fail argument to
uint32'-shiftr out of bounds else Uint32.shiftr w i)

```

**code-printing constant** *uint32-sshiftr*  $\rightarrow$

```

(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 w => fn i => if i < 0 orelse i >= 32 then raise Fail argument to
  uint32'-shiftr'-signed out of bounds else Uint32.shiftr'-signed w i)
```

**context**

**includes** *bit-operations-syntax*

**begin**

**lemma** *uint32-msb-test-bit*:  $msb\ x \longleftrightarrow bit\ (x :: uint32)\ 31$   
 ⟨*proof*⟩

**lemma** *msb-uint32-code* [code]:  $msb\ x \longleftrightarrow uint32-test-bit\ x\ 31$   
 ⟨*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]:  
 $uint32-of-nat = uint32-of-int \circ int$   
 ⟨*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* ::  $uint32 \Rightarrow integer$   
**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  $\rightarrow$ 
  (SML) IntInf.fromLarge (Word32.toLargeInt -) : IntInf.int and
  (Haskell) Prelude.toInteger
| constant integer-of-uint32-signed  $\rightarrow$ 
  (OCaml) Z.of'-int32 and
  (Scala) BigInt

```

### 4.3 Quickcheck setup

**definition** *uint32-of-natural* :: *natural*  $\Rightarrow$  *uint32*  
**where** *uint32-of-natural* *x*  $\equiv$  *Uint32* (*integer-of-natural* *x*)

**instantiation** *uint32* :: {*random*, *exhaustive*, *full-exhaustive*} **begin**

**definition** *random-uint32*  $\equiv$  *qc-random-cnv uint32-of-natural*

**definition** *exhaustive-uint32*  $\equiv$  *qc-exhaustive-cnv uint32-of-natural*

**definition** *full-exhaustive-uint32*  $\equiv$  *qc-full-exhaustive-cnv uint32-of-natural*

**instance**  $\langle$ *proof* $\rangle$

end

**instantiation** *uint32* :: *narrowing* **begin**

**interpretation** *quickcheck-narrowing-samples*

$\lambda i.$  let *x* = *Uint32 i* in (*x*, *0xFFFFFFFF - x*) 0

*Typerep.Typerep* (*STR "Uint32.uint32"*) []  $\langle$ *proof* $\rangle$

**definition** *narrowing-uint32* *d* = *qc-narrowing-drawn-from* (*narrowing-samples* *d*)

**lemmas** *partial-term-of-uint32* [*code*] = *partial-term-of-code*

**instance**  $\langle$ *proof* $\rangle$

end

end

## Chapter 5

# Unsigned words of 16 bits

```
theory Uint16
imports
  Uint-Common
  Code-Target-Word
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*.

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

### 5.1 Type definition and primitive operations

```
typedef uint16 =  $\langle UNIV :: 16 \text{ word set} \rangle \langle \text{proof} \rangle$ 
```

```
global-interpretation uint16: word-type-copy Abs-uint16 Rep-uint16
   $\langle \text{proof} \rangle$ 
```

```
setup-lifting type-definition-uint16
```

```
declare uint16.of-word-of [code abstype]
```

```
declare Quotient-uint16 [transfer-rule]
```

```
instantiation uint16 ::  $\langle \{ \text{comm-ring-1, semiring-modulo, equal, linorder, order-bot,} \rangle$ 
   $\langle \text{order-top} \rangle$ 
```

```
begin
```

```
lift-definition zero-uint16 :: uint16 is 0  $\langle \text{proof} \rangle$ 
```

```

lift-definition one-uint16 :: uint16 is 1 ⟨proof⟩
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⟩
lift-definition bot-uint16 :: uint16 is bot ⟨proof⟩
lift-definition top-uint16 :: uint16 is top ⟨proof⟩

```

```

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

```

```

instance ⟨proof⟩

```

```

end

```

```

instance uint16 :: ⟨{interval-bot, interval-top}⟩
  ⟨proof⟩

```

```

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 :: ⟨uint16 ⇒ nat⟩
  is unat ⟨proof⟩

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

lift-definition int-of-uint16 :: ⟨uint16 ⇒ int⟩
  is uint ⟨proof⟩

context
  includes integer.lifting
begin

lift-definition Uint16 :: ⟨integer ⇒ uint16⟩
  is word-of-int ⟨proof⟩

lift-definition integer-of-uint16 :: ⟨uint16 ⇒ integer⟩
  is uint ⟨proof⟩

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
  ⟨proof⟩

instantiation uint16 :: {size, msb, bit-comprehension}
begin

lift-definition size-uint16 :: ⟨uint16 ⇒ nat⟩ is size ⟨proof⟩

lift-definition msb-uint16 :: ⟨uint16 ⇒ bool⟩ is msb ⟨proof⟩

lift-definition set-bits-uint16 :: ⟨(nat ⇒ bool) ⇒ uint16⟩ is set-bits ⟨proof⟩
lift-definition set-bits-aux-uint16 :: ⟨(nat ⇒ bool) ⇒ nat ⇒ uint16 ⇒ uint16⟩ is
  set-bits-aux ⟨proof⟩

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
  ⟨proof⟩

instance ⟨proof⟩

```

end

## 5.2 Code setup

```

code-printing code-module Uint16  $\rightarrow$  (SML-word)
⟨(* 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 shiftl : 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 shiftl 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)))
  <> Word16.fromInt 0

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

code-printing code-module Uint16  $\rightarrow$  (Haskell)
⟨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)
⟨object Uint16 {

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 (=) (λx. x) Rep-uint16'*  
 ⟨proof⟩

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

**lift-definition** *Abs-uint16' :: 16 word ⇒ uint16 is λx :: 16 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 ⇒ -*]]

**lemma** *term-of-uint16-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 "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 i)*  
 ⟨proof⟩

**code-printing**

**type-constructor** *uint16* ↯  
 (*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  $\Rightarrow$  -  $\rightarrow$ 
  (SML-word) Word16.- ((-), (-)) and
  (Haskell) infixl 6 - and
  (Scala) (- -/ -).toChar
| constant times :: uint16  $\Rightarrow$  -  $\Rightarrow$  -  $\rightarrow$ 
  (SML-word) Word16.* ((-), (-)) and
  (Haskell) infixl 7 * and
  (Scala) (- */ -).toChar
| constant HOL.equal :: uint16  $\Rightarrow$  -  $\Rightarrow$  bool  $\rightarrow$ 
  (SML-word) !((- : Word16.word) = -) and
  (Haskell) infix 4 == and
  (Scala) infixl 5 ==
| class-instance uint16 :: equal  $\rightarrow$  (Haskell) -
| constant less-eq :: uint16  $\Rightarrow$  -  $\Rightarrow$  bool  $\rightarrow$ 
  (SML-word) Word16.<= ((-), (-)) and
  (Haskell) infix 4 <= and
  (Scala) infixl 4 <=
| constant less :: uint16  $\Rightarrow$  -  $\Rightarrow$  bool  $\rightarrow$ 
  (SML-word) Word16.< ((-), (-)) and
  (Haskell) infix 4 < and
  (Scala) infixl 4 <
| constant Bit-Operations.not :: uint16  $\Rightarrow$  -  $\rightarrow$ 
  (SML-word) Word16.notb and
  (Haskell) Data'-Bits.complement and
  (Scala) -.unary'-~.toChar
| constant Bit-Operations.and :: uint16  $\Rightarrow$  -  $\rightarrow$ 

```

```

(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 mod y* = (if *y* = 0 then *x* else *uint16-mod* *x y*)  
 ⟨*proof*⟩

**lemma** *uint16-div-code* [*code*]:  
*Rep-uint16* (*uint16-div* *x y*) =  
 (if *y* = 0 then *Rep-uint16* (undefined ((*div*) :: *uint16* ⇒ -) *x* (0 :: *uint16*)) else  
*Rep-uint16* *x div Rep-uint16 y*)  
 ⟨*proof*⟩

**lemma** *uint16-mod-code* [*code*]:  
*Rep-uint16* (*uint16-mod* *x y*) =  
 (if *y* = 0 then *Rep-uint16* (undefined ((*mod*) :: *uint16* ⇒ -) *x* (0 :: *uint16*)) else  
*Rep-uint16* *x mod Rep-uint16 y*)  
 ⟨*proof*⟩

**end**

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

(Scala) (- % -).toChar

**global-interpretation** *uint16*: word-type-copy-target-language 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 16 15*

**defines** *uint16-test-bit* = *uint16.test-bit*

**and** *uint16-shiffl* = *uint16.shiffl*

**and** *uint16-shiftr* = *uint16.shiftr*

**and** *uint16-sshiftr* = *uint16.sshiftr*

*<proof>*

**code-printing constant** *uint16-test-bit*  $\rightarrow$

(SML-word) *Uint16.test'-bit* **and**

(Haskell) *Data'-Bits.testBitBounded* **and**

(Scala) *Uint16.test'-bit*

**code-printing constant** *uint16-shiffl*  $\rightarrow$

(SML-word) *Uint16.shiffl* **and**

(Haskell) *Data'-Bits.shifflBounded* **and**

(Scala) *Uint16.shiffl*

**code-printing constant** *uint16-shiftr*  $\rightarrow$

(SML-word) *Uint16.shiftr* **and**

(Haskell) *Data'-Bits.shiftrBounded* **and**

(Scala) *Uint16.shiftr*

**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*:  $msb\ x \longleftrightarrow bit\ (x :: uint16)\ 15$

*<proof>*

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

*<proof>*

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

**including** *integer.lifting* *<proof>*

**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*

### 5.3 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*)

*d*

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

**instance** *<proof>*

**end**

**end**



## Chapter 6

# Unsigned words of 8 bits

```
theory Uint8
  imports
    Uint-Common
    Code-Target-Word
begin
```

Restriction for OCaml code generation: OCaml does not provide an `int8` type, so no special code generation for this type is set up.

### 6.1 Type definition and primitive operations

```
typedef uint8 = ⟨UNIV :: 8 word set⟩ ⟨proof⟩

global-interpretation uint8: word-type-copy Abs-uint8 Rep-uint8
  ⟨proof⟩

setup-lifting type-definition-uint8

declare uint8.of-word-of [code abstype]

declare Quotient-uint8 [transfer-rule]

instantiation uint8 :: ⟨{comm-ring-1, semiring-modulo, equal, linorder, order-bot,
order-top}⟩
begin

lift-definition zero-uint8 :: uint8 is 0 ⟨proof⟩
lift-definition one-uint8 :: uint8 is 1 ⟨proof⟩
lift-definition plus-uint8 :: ⟨uint8 ⇒ uint8 ⇒ uint8⟩ is ⟨(+)⟩ ⟨proof⟩
lift-definition uminus-uint8 :: ⟨uint8 ⇒ uint8⟩ is uminus ⟨proof⟩
lift-definition minus-uint8 :: ⟨uint8 ⇒ uint8 ⇒ uint8⟩ is ⟨(-)⟩ ⟨proof⟩
lift-definition times-uint8 :: ⟨uint8 ⇒ uint8 ⇒ uint8⟩ is ⟨(*)⟩ ⟨proof⟩
lift-definition divide-uint8 :: ⟨uint8 ⇒ uint8 ⇒ uint8⟩ is ⟨(div)⟩ ⟨proof⟩
lift-definition modulo-uint8 :: ⟨uint8 ⇒ uint8 ⇒ uint8⟩ is ⟨(mod)⟩ ⟨proof⟩
```

**lift-definition** *equal-uint8* ::  $\langle uint8 \Rightarrow uint8 \Rightarrow bool \rangle$  **is**  $\langle HOL.equal \rangle$   $\langle proof \rangle$

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

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

**lift-definition** *bot-uint8* :: *uint8* **is** *bot*  $\langle proof \rangle$

**lift-definition** *top-uint8* :: *uint8* **is** *top*  $\langle proof \rangle$

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

**instance**  $\langle proof \rangle$

**end**

**instance** *uint8* ::  $\langle \{interval-bot, interval-top\} \rangle$   
 $\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**instance**  
 $\langle proof \rangle$

**end**

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

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

```

lift-definition uint8-of-int ::  $\langle int \Rightarrow uint8 \rangle$ 
  is word-of-int  $\langle proof \rangle$ 

lift-definition int-of-uint8 ::  $\langle uint8 \Rightarrow int \rangle$ 
  is uint  $\langle proof \rangle$ 

context
  includes integer.lifting
begin

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

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

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
   $\langle proof \rangle$ 

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

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

lift-definition msb-uint8 ::  $\langle uint8 \Rightarrow bool \rangle$  is msb  $\langle proof \rangle$ 

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

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
   $\langle proof \rangle$ 

instance  $\langle proof \rangle$ 

end

```

## 6.2 Code setup

```

code-printing code-module Uint8  $\rightarrow$  (SML)
 $\langle (*$  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 shiftl : Word8.word -> IntInf.int -> Word8.word
    val shiftr : Word8.word -> IntInf.int -> Word8.word
    val shiftr-signed : Word8.word -> IntInf.int -> Word8.word
    val test-bit : Word8.word -> IntInf.int -> bool
end = struct

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 -> (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 -> (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 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 *s* 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'* :: *s* word  $\Rightarrow$  *uint8* **is**  $\lambda x :: s \text{ word}. x$  ⟨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*  $\Rightarrow$  -]]

**lemma** *term-of-uint8-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 "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]:  $Rep-uint8 (Uint8 i) = word-of-int (int-of-integer i)$

```
⟨proof⟩
```

**code-printing type-constructor** *uint8*  $\rightarrow$

```
(SML) Word8.word and
```

```

(Haskell) Uint8.Word8 and
(Scala) Byte
| constant Uint8  $\rightarrow$ 
(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  $\rightarrow$ 
(SML) (Word8.fromInt 0) and
(Haskell) (0 :: Uint8.Word8) and
(Scala) 0.toByte
| constant 1 :: uint8  $\rightarrow$ 
(SML) (Word8.fromInt 1) and
(Haskell) (1 :: Uint8.Word8) and
(Scala) 1.toByte
| constant plus :: uint8  $\Rightarrow$  -  $\Rightarrow$  -  $\rightarrow$ 
(SML) Word8.+ ((-), (-)) and
(Haskell) infixl 6 + and
(Scala) (- +/ -).toByte
| constant uminus :: uint8  $\Rightarrow$  -  $\rightarrow$ 
(SML) Word8.~ and
(Haskell) negate and
(Scala) (- -).toByte
| constant minus :: uint8  $\Rightarrow$  -  $\rightarrow$ 
(SML) Word8.- ((-), (-)) and
(Haskell) infixl 6 - and
(Scala) (- -/ -).toByte
| constant times :: uint8  $\Rightarrow$  -  $\Rightarrow$  -  $\rightarrow$ 
(SML) Word8.* ((-), (-)) and
(Haskell) infixl 7 * and
(Scala) (- */ -).toByte
| constant HOL.equal :: uint8  $\Rightarrow$  -  $\Rightarrow$  bool  $\rightarrow$ 
(SML) !((- : Word8.word) = -) and
(Haskell) infix 4 == and
(Scala) infixl 5 ==
| class-instance uint8 :: equal  $\rightarrow$  (Haskell) -
| constant less-eq :: uint8  $\Rightarrow$  -  $\Rightarrow$  bool  $\rightarrow$ 
(SML) Word8.<= ((-), (-)) and
(Haskell) infix 4 <= and
(Scala) Uint8.less'-eq
| constant less :: uint8  $\Rightarrow$  -  $\Rightarrow$  bool  $\rightarrow$ 
(SML) Word8.< ((-), (-)) and
(Haskell) infix 4 < and
(Scala) Uint8.less
| constant Bit-Operations.not :: uint8  $\Rightarrow$  -  $\rightarrow$ 
(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 div y = (if y = 0 then 0 else *uint8-div* x y)  
**including** *undefined-transfer* ⟨proof⟩

**lemma** *mod-uint8-code* [code]: x mod y = (if y = 0 then x else *uint8-mod* x y)  
**including** *undefined-transfer* ⟨proof⟩

**definition** *uint8-sdiv* :: uint8 ⇒ uint8 ⇒ uint8  
**where**

*uint8-sdiv* x y =  
 (if y = 0 then undefined ((div) :: uint8 ⇒ -) x (0 :: uint8)  
 else *Abs-uint8* (*Rep-uint8* x *sdiv* *Rep-uint8* y))

**definition** *div0-uint8* :: uint8 ⇒ uint8  
**where** [code abort]: *div0-uint8* x = undefined ((div) :: uint8 ⇒ -) x (0 :: uint8)

**definition** *mod0-uint8* :: uint8 ⇒ uint8  
**where** [code abort]: *mod0-uint8* x = undefined ((mod) :: uint8 ⇒ -) x (0 :: uint8)

**lemma** *uint8-divmod-code* [code]:  
*uint8-divmod* x y =  
 (if 0x80 ≤ y then if x < y then (0, x) else (1, x - y)  
 else if y = 0 then (*div0-uint8* x, *mod0-uint8* x)  
 else let q = *push-bit* 1 (*uint8-sdiv* (*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 (uint8-sdiv x y) =*  
*(if y = 0 then Rep-uint8 (undefined ((div) :: uint8 => -) x (0 :: uint8))*  
*else Rep-uint8 x sdiv Rep-uint8 y)*  
*<proof>*

Note that we only need a translation for signed division, but not for the remainder because *uint8-divmod ?x ?y = (if 128 ≤ ?y then if ?x < ?y then (0, ?x) else (1, ?x - ?y) else if ?y = 0 then (div0-uint8 ?x, mod0-uint8 ?x) else let q = push-bit 1 (uint8-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** *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*

**global-interpretation** *uint8: word-type-copy-target-language 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 8 7*

**defines** *uint8-test-bit = uint8.test-bit*  
**and** *uint8-shiffl = uint8.shiffl*  
**and** *uint8-shiftr = uint8.shiftr*  
**and** *uint8-sshiftr = uint8.sshiftr*  
*<proof>*

**code-printing constant** *uint8-test-bit*  $\rightarrow$

*(SML) Uint8.test'-bit and*  
*(Haskell) Data'-Bits.testBitBounded and*  
*(Scala) Uint8.test'-bit and*  
*(Eval) (fn w => fn i => if i < 0 orelse i >= 8 then raise (Fail argument to uint8'-test'-bit out of bounds) else Uint8.test'-bit w i)*

**code-printing constant** *uint8-shiffl*  $\rightarrow$

*(SML) Uint8.shiffl and*  
*(Haskell) Data'-Bits.shifflBounded and*  
*(Scala) Uint8.shiffl and*  
*(Eval) (fn w => fn i => if i < 0 orelse i >= 8 then raise (Fail argument to*

*uint8'-shiftr out of bounds*) else *Uin8.shiftr w i*)

**code-printing constant** *uint8-shiftr*  $\rightarrow$

(SML) *Uin8.shiftr* **and**

(Haskell) *Data'-Bits.shiftrBounded* **and**

(Scala) *Uin8.shiftr* **and**

(Eval) (fn w => fn i => if i < 0 orelse i >= 8 then raise (Fail argument to *uint8'-shiftr out of bounds*) else *Uin8.shiftr w i*)

**code-printing constant** *uint8-sshiftr*  $\rightarrow$

(SML) *Uin8.shiftr'-signed* **and**

(Haskell)

(Prelude.fromInteger (Prelude.toInteger (Data'-Bits.shiftrBounded (Prelude.fromInteger (Prelude.toInteger -) :: Uin8.Int8) -)) :: Uin8.Word8) **and**

(Scala) *Uin8.shiftr'-signed* **and**

(Eval) (fn w => fn i => if i < 0 orelse i >= 8 then raise (Fail argument to *uint8'-sshiftr out of bounds*) else *Uin8.shiftr'-signed w 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 = Uin8 (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 o 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*  $\Rightarrow$  *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-wint8-signed-code [code]:
  integer-of-wint8-signed n =
    (if bit n 7 then undefined integer-of-wint8 n else integer-of-int (uint (Rep-wint8'
n)))
  ⟨proof⟩

lemma integer-of-wint8-code [code]:
  integer-of-wint8 n =
    (if bit n 7 then integer-of-wint8-signed (n AND 0x7F) OR 0x80 else integer-of-wint8-signed
n)
  ⟨proof⟩
  including integer.lifting ⟨proof⟩

```

end

### code-printing

```

constant integer-of-wint8 →
  (SML) IntInf.fromLarge (Word8.toLargeInt -) and
  (Haskell) Prelude.toInteger
| constant integer-of-wint8-signed →
  (Scala) BigInt

```

## 6.3 Quickcheck setup

```

definition wint8-of-natural :: natural ⇒ wint8
where wint8-of-natural x ≡ Uint8 (integer-of-natural x)

```

```

instantiation wint8 :: {random, exhaustive, full-exhaustive} begin
definition random-wint8 ≡ qc-random-cnv wint8-of-natural
definition exhaustive-wint8 ≡ qc-exhaustive-cnv wint8-of-natural
definition full-exhaustive-wint8 ≡ qc-full-exhaustive-cnv wint8-of-natural
instance ⟨proof⟩
end

```

```

instantiation wint8 :: narrowing begin

```

```

interpretation quickcheck-narrowing-samples
  λi. let x = Uint8 i in (x, 0xFF - x) 0
  Typerep.Typerep (STR "Uint8.wint8") [] ⟨proof⟩

```

```

definition narrowing-wint8 d = qc-narrowing-drawn-from (narrowing-samples d)
d

```

```

lemmas partial-term-of-wint8 [code] = partial-term-of-code

```

```

instance ⟨proof⟩
end

```

end

## Chapter 7

# Unsigned words of default size

```
theory Uint
  imports
    Uint-Common
    Code-Target-Word
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'")  $\square$ 
```

```
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**  $dflt\text{-}size \equiv len\text{-}of (TYPE (dflt\text{-}size))$

**context includes** *integer.lifting* **begin**

**lift-definition**  $dflt\text{-}size\text{-}integer :: integer \text{ is } int\ dflt\text{-}size \langle proof \rangle$

**declare**  $dflt\text{-}size\text{-}integer\text{-}def$  [code drop]

— The code generator will substitute a machine-dependent value for this constant

**lemma**  $dflt\text{-}size\text{-}by\text{-}int$  [code]:  $dflt\text{-}size = nat\text{-}of\text{-}integer\ dflt\text{-}size\text{-}integer$   
 $\langle proof \rangle$

**lemma**  $dflt\text{-}size$  [simp]:

$dflt\text{-}size > 0$

$dflt\text{-}size \geq Suc\ 0$

$\neg dflt\text{-}size < Suc\ 0$

$\langle proof \rangle$

**end**

## 7.1 Type definition and primitive operations

**typedef**  $uint = \langle UNIV :: dflt\text{-}size\ \text{word}\ \text{set} \rangle \langle proof \rangle$

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

**setup-lifting**  $type\text{-}definition\text{-}uint$

**declare**  $uint.of\text{-}word\text{-}of$  [code abstype]

**declare**  $Quotient\text{-}uint$  [transfer-rule]

**instantiation**  $uint :: \langle \{ comm\text{-}ring\text{-}1, semiring\text{-}modulo, equal, linorder, order\text{-}bot, order\text{-}top \} \rangle$

**begin**

**lift-definition**  $zero\text{-}uint :: uint \text{ is } 0 \langle proof \rangle$

**lift-definition**  $one\text{-}uint :: uint \text{ is } 1 \langle proof \rangle$

**lift-definition**  $plus\text{-}uint :: \langle uint \Rightarrow uint \Rightarrow uint \rangle \text{ is } \langle (+) \rangle \langle proof \rangle$

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

**lift-definition**  $minus\text{-}uint :: \langle uint \Rightarrow uint \Rightarrow uint \rangle \text{ is } \langle (-) \rangle \langle proof \rangle$

**lift-definition**  $times\text{-}uint :: \langle uint \Rightarrow uint \Rightarrow uint \rangle \text{ is } \langle (*) \rangle \langle proof \rangle$

**lift-definition**  $divide\text{-}uint :: \langle uint \Rightarrow uint \Rightarrow uint \rangle \text{ is } \langle (div) \rangle \langle proof \rangle$

**lift-definition**  $modulo\text{-}uint :: \langle uint \Rightarrow uint \Rightarrow uint \rangle \text{ is } \langle (mod) \rangle \langle proof \rangle$

**lift-definition**  $equal\text{-}uint :: \langle uint \Rightarrow uint \Rightarrow bool \rangle \text{ is } \langle HOL.equal \rangle \langle proof \rangle$

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

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

**lift-definition**  $bot\text{-}uint :: uint \text{ is } bot \langle proof \rangle$

```

lift-definition top-uint :: uint is top ⟨proof⟩

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

instance ⟨proof⟩

end

instance uint :: ⟨{interval-bot, interval-top}⟩
  ⟨proof⟩

instantiation uint :: ring-bit-operations
begin

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

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

instance
  ⟨proof⟩

end

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

lift-definition nat-of-uint :: ⟨uint ⇒ nat⟩
  is unat ⟨proof⟩

lift-definition uint-of-int :: ⟨int ⇒ uint⟩
  is word-of-int ⟨proof⟩

lift-definition int-of-uint :: ⟨uint ⇒ int⟩
  is uint ⟨proof⟩

```

```

context
  includes integer.lifting
begin

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

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 :: {size, msb, 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 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

```

## 7.2 Code setup

```

code-printing code-module Uint  $\rightarrow$  (SML)
<
structure Uint : sig
  val shiftl : Word.word  $\rightarrow$  IntInf.int  $\rightarrow$  Word.word
  val shiftr : Word.word  $\rightarrow$  IntInf.int  $\rightarrow$  Word.word
  val shiftr-signed : Word.word  $\rightarrow$  IntInf.int  $\rightarrow$  Word.word
  val test-bit : Word.word  $\rightarrow$  IntInf.int  $\rightarrow$  bool
end = struct

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

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

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

def test-bit(x: Int, n: BigInt) : Boolean =
  (x & (1 << n.intValue)) != 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* and *bit-operations-syntax*  
**begin**

**definition** *wivs-mask* :: *int* **where** *wivs-mask* =  $2^{\text{dflt-size} - 1}$

**lift-definition** *wivs-mask-integer* :: *integer* **is** *wivs-mask* *<proof>*

**lemma** [*code*]: *wivs-mask-integer* =  $2^{\text{dflt-size} - 1}$   
*<proof>*

**definition** *wivs-shift* :: *int* **where** *wivs-shift* =  $2^{\text{dflt-size}}$

**lift-definition** *wivs-shift-integer* :: *integer* **is** *wivs-shift* *<proof>*

**lemma** [*code*]: *wivs-shift-integer* =  $2^{\text{dflt-size}}$   
*<proof>*

**definition** *wivs-index* :: *nat* **where** *wivs-index* ==  $\text{dflt-size} - 1$

**lift-definition** *wivs-index-integer* :: *integer* **is** *int wivs-index* *<proof>*

**lemma** *wivs-index-integer-code*[*code*]: *wivs-index-integer* = *dflt-size-integer* - 1  
*<proof>*

**definition** *wivs-overflow* :: *int* **where** *wivs-overflow* ==  $2^{(\text{dflt-size} - 1)}$

**lift-definition** *wivs-overflow-integer* :: *integer* **is** *wivs-overflow* *<proof>*

**lemma** [*code*]: *wivs-overflow-integer* =  $2^{(\text{dflt-size} - 1)}$

*<proof>*

**definition** *wivs-least* :: *int* **where** *wivs-least* == - *wivs-overflow*

**lift-definition** *wivs-least-integer* :: *integer* **is** *wivs-least* *<proof>*

**lemma** [*code*]: *wivs-least-integer* = - (2 <sup>^(dflt-size - 1)</sup>)

*<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* *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*  $\Rightarrow$  *uint* **is**  $\lambda x ::$  *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*  $\Rightarrow$  -]]

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

**defines** *TR*  $\equiv$  *typerep.Type* **and** *bit0*  $\equiv$  *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.dflt'-size and
  (OCaml) Uint.dflt'-size and
  (Scala) Uint.dflt'-size
| constant Uint →
  (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 →
  (OCaml) Z.to'-int
| constant 0 :: uint →
  (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 →
  (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 ⇒ - →
  (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 ⇒ - →
  (SML) Word.~ and
  (Eval) (raise (Fail Machine dependent code)) and
  (Quickcheck) Word.~ and
  (Haskell) negate and
  (OCaml) Pervasives.(~ -) and
  (Scala) !(- -)
| constant minus :: uint ⇒ - →
  (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 ⇒ - →
(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 ⇒ uint ⇒ uint × uint where
  uint-divmod x y =
    (if y = 0 then (undefined ((div) :: uint ⇒ -) x (0 :: uint), undefined ((mod) ::
uint ⇒ -) x (0 :: uint))
    else (x div y, x mod y))

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

definition uint-mod :: uint ⇒ uint ⇒ 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 ⟨proof⟩

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

```

**definition** *uint-sdiv* :: *uint*  $\Rightarrow$  *uint*  $\Rightarrow$  *uint*

**where**

*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 abort]: *div0-uint* *x* = undefined ((*div*) :: *uint*  $\Rightarrow$  -) *x* (0 :: *uint*)

**definition** *mod0-uint* :: *uint*  $\Rightarrow$  *uint*

**where** [code abort]: *mod0-uint* *x* = undefined ((*mod*) :: *uint*  $\Rightarrow$  -) *x* (0 :: *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*:

$\langle$ *wivs-overflow-uint* > 0 $\rangle$   
 $\langle$ proof $\rangle$

**lemma** *uint-divmod-code* [code]:

*uint-divmod* *x y* =  
 (if *wivs-overflow-uint*  $\leq$  *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*  $\geq$  *y* then (*q* + 1, *r* - *y*) else (*q*, *r*)

$\langle$ proof $\rangle$

**including** *undefined-transfer*

$\langle$ proof $\rangle$

**lemma** *uint-sdiv-code* [code]:

Rep-uint (uint-sdiv *x y*) =  
 (if *y* = 0 then Rep-uint (undefined ((*div*) :: *uint*  $\Rightarrow$  -) *x* (0 :: *uint*)  
 else Rep-uint *x* sdiv Rep-uint *y*)

$\langle$ proof $\rangle$

Note that we only need a translation for signed division, but not for the remainder because *uint-divmod* ?*x* ?*y* = (if *wivs-overflow-uint*  $\leq$  ?*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*  $\leq$  *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) - '/' -

```

**global-interpretation** *uint: word-type-copy-target-language 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*  
*<of-nat dflt-size> wivs-index*  
**defines** *uint-test-bit = uint.test-bit*  
   **and** *uint-shiffl = uint.shiffl*  
   **and** *uint-shiftr = uint.shiftr*  
   **and** *uint-sshiftr = uint.sshiftr*  
*<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*

**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*

**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*

**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*:  $msb\ x \longleftrightarrow bit\ (x :: uint)\ wivs-index$   
 $\langle proof \rangle$

**lemma** *msb-uint-code* [code]:  $msb\ x \longleftrightarrow uint-test-bit\ x\ wivs-index-integer$   
 $\langle proof \rangle$

**lemma** *uint-of-int-code* [code]:  $uint-of-int\ i = (BITS\ n.\ bit\ i\ n)$   
 $\langle proof \rangle$

### 7.3 Quickcheck setup

**definition** *uint-of-natural* ::  $natural \Rightarrow uint$   
**where** *uint-of-natural*  $x \equiv Uint\ (integer-of-natural\ x)$

**instantiation** *uint* :: {random, exhaustive, full-exhaustive} **begin**  
**definition** *random-uint*  $\equiv qc-random-cnv\ uint-of-natural$   
**definition** *exhaustive-uint*  $\equiv qc-exhaustive-cnv\ uint-of-natural$   
**definition** *full-exhaustive-uint*  $\equiv qc-full-exhaustive-cnv\ uint-of-natural$   
**instance**  $\langle proof \rangle$   
**end**

**instantiation** *uint* :: *narrowing* **begin**

**interpretation** *quickcheck-narrowing-samples*  
 $\lambda i.\ (Uint\ i,\ Uint\ (-\ i))\ 0$   
*Typerep.Typerep* (STR "Uint.uint") []  $\langle proof \rangle$

**definition** *narrowing-uint*  $d = qc-narrowing-drawn-from\ (narrowing-samples\ d)\ d$

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

**instance**  $\langle proof \rangle$   
**end**

**end**

## Chapter 8

# 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>*

### 8.1 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*  $x = \text{Abs-uint8}' (\text{ucast} (\text{Rep-uint}' x))$   
 ⟨proof⟩

**lemma** *uint16-of-uint-code* [code]:  
*uint16-of-uint*  $x = \text{Abs-uint16}' (\text{ucast} (\text{Rep-uint}' x))$   
 ⟨proof⟩

**lemma** *uint32-of-uint-code* [code]:  
*uint32-of-uint*  $x = \text{Abs-uint32}' (\text{ucast} (\text{Rep-uint}' x))$   
 ⟨proof⟩

**lemma** *uint64-of-uint-code* [code]:  
*uint64-of-uint*  $x = \text{Abs-uint64}' (\text{ucast} (\text{Rep-uint}' x))$

*<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**

## 8.2 Compatibility with Imperative/HOL

**theory** *Native-Word-Imperative-HOL* **imports**

*Code-Target-Word*

*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 9

## Test cases

```
theory Native-Word-Test
imports
  Uint64 Uint32 Uint16 Uint8 Uint Native-Cast-Uint
  HOL-Library.Code-Test
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
```

### 9.1 Tests for $\text{isa}^{\wedge}\text{typ}$ *integer*

```
context
  includes bit-operations-syntax
begin

definition bit-integer-test :: bool
  where  $\langle \text{bit-integer-test} =$ 
  (([ -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)
    , Bit-Operations.set-bit 4 5, Bit-Operations.set-bit 2 (- 5)
    , Bit-Operations.unset-bit 0 5, Bit-Operations.unset-bit 1 (- 5)
    , Bit-Operations.flip-bit 4 5, Bit-Operations.flip-bit 1 (- 5)
    , push-bit 2 1, push-bit 3 (- 1)
    , drop-bit 3 100, drop-bit 3 (- 100)
    , take-bit 4 100, take-bit 4 (- 100)] :: integer list)
```

```

= [ 3, 1, 1, -7
  , -3, -3, 7, -1
  , -2, 2
  , -4, -4, 6, 6
  , 21, -1, 4, -7, 21, -7
  , 4, -8
  , 12, -13
  , 4, 12] ^
[ bit (5 :: integer) 4, bit (5 :: integer) 2, bit (-5 :: integer) 4, bit (-5 ::
integer) 2
  , bit (5 :: integer) 0, bit (4 :: integer) 0, bit (-1 :: integer) 0, bit (-2 ::
integer) 0,
  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 AND y = x OR (y :: integer)⟩
quickcheck [random, expect=counterexample]
quickcheck [exhaustive, expect=counterexample]
⟨proof⟩

```

```

lemma ⟨(x :: integer) AND x = x OR x⟩
quickcheck [narrowing, expect=no-counterexample]
⟨proof⟩

```

```

lemma ⟨(f :: integer ⇒ unit) = g⟩
quickcheck[narrowing, size=3, expect=no-counterexample]
⟨proof⟩

```

```

end

```

## 9.2 Tests for *uint8*

```

context
  includes bit-operations-syntax
begin

```

```

definition test-uint8 :: bool

```

```

where <test-uint8 <→>
(( [ 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
  , Bit-Operations.set-bit 4 5, Bit-Operations.set-bit 2 (- 5)
  , Bit-Operations.set-bit 32 5, Bit-Operations.set-bit 32 (- 5)
  , Bit-Operations.unset-bit 0 5, Bit-Operations.unset-bit 1 (- 5)
  , Bit-Operations.unset-bit 32 5, Bit-Operations.unset-bit 32 (- 5)
  , Bit-Operations.flip-bit 4 5, Bit-Operations.flip-bit 1 (- 5)
  , Bit-Operations.flip-bit 32 5, Bit-Operations.flip-bit 32 (- 5)
  , push-bit 2 1, push-bit 3 (- 1), push-bit 8 1, push-bit 0 1
  , drop-bit 3 100, drop-bit 3 (- 100), drop-bit 8 100, drop-bit 8 (- 100)
  , signed-drop-bit-uint8 3 100, signed-drop-bit-uint8 3 (- 100)
  , signed-drop-bit-uint8 8 100, signed-drop-bit-uint8 8 (- 100)
  , take-bit 4 100, take-bit 4 (- 100)] :: 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, 5, 251, 4, 249, 5, 251, 21, 249, 5, 251
  , 4, 248, 0, 1
  , 12, 19, 0, 0
  , 12, 243, 0, 255
  , 4, 12] ) ^
(( (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' = drop-bit 2 (push-bit 3 (0 + 10 - 14 * 3 div 6 mod 3))⟩

```

⟨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⟩

```

### 9.3 Tests for *uint16*

```

context
  includes bit-operations-syntax
begin

```

```

definition test-uint16 :: bool
  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
, Bit-Operations.set-bit 4 5, Bit-Operations.set-bit 2 (- 5)
, Bit-Operations.set-bit 32 5, Bit-Operations.set-bit 32 (- 5)
, Bit-Operations.unset-bit 0 5, Bit-Operations.unset-bit 1 (- 5)
, Bit-Operations.unset-bit 32 5, Bit-Operations.unset-bit 32 (- 5)
, Bit-Operations.flip-bit 4 5, Bit-Operations.flip-bit 1 (- 5)
, Bit-Operations.flip-bit 32 5, Bit-Operations.flip-bit 32 (- 5)
, push-bit 2 1, push-bit 3 (- 1), push-bit 16 1, push-bit 0 1
, drop-bit 3 100, drop-bit 3 (- 100), drop-bit 16 100, drop-bit 16 (- 100)
, signed-drop-bit-uint16 3 100, signed-drop-bit-uint16 3 (- 100)
, signed-drop-bit-uint16 16 100, signed-drop-bit-uint16 16 (- 100)
, take-bit 4 100, take-bit 4 (- 100)] :: 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, 5, 65531, 4, 65529, 5, 65531, 21, 65529, 5, 65531
, 4, 65528, 0, 1
, 12, 8179, 0, 0
, 12, 65523, 0, 65535
, 4, 12]) ^
([ (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

```

## 9.4 Tests for *uint32*

```

context
  includes bit-operations-syntax
begin

```

```

definition test-uint32 :: bool
  where <test-uint32 <=>
  (([ 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 div 3, -5 div 3, -5 div -3, 5 div -3
    , 5 mod 3, -5 mod 3, -5 mod -3, 5 mod -3
    , Bit-Operations.set-bit 4 5, Bit-Operations.set-bit 2 (- 5)
    , Bit-Operations.set-bit 32 5, Bit-Operations.set-bit 32 (- 5)
    , Bit-Operations.unset-bit 0 5, Bit-Operations.unset-bit 1 (- 5)
    , Bit-Operations.unset-bit 32 5, Bit-Operations.unset-bit 32 (- 5)
    , Bit-Operations.flip-bit 4 5, Bit-Operations.flip-bit 1 (- 5)
    , Bit-Operations.flip-bit 32 5, Bit-Operations.flip-bit 32 (- 5)
    , push-bit 2 1, push-bit 3 (- 1), push-bit 32 1, push-bit 0 1

```

```

, drop-bit 3 100, drop-bit 3 (- 100), drop-bit 32 100, drop-bit 32 (- 100)
, signed-drop-bit-uint32 3 100, signed-drop-bit-uint32 3 (- 100)
, signed-drop-bit-uint32 32 100, signed-drop-bit-uint32 32 (- 100)
, take-bit 4 100, take-bit 4 (- 100)] :: 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, 5, 4294967291, 4, 4294967289, 5, 4294967291, 21, 4294967289,
5, 4294967291
, 4, 4294967288, 0, 1
, 12, 536870899, 0, 0
, 12, 4294967283, 0, 4294967295
, 4, 12]) ^
([ (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
]
)
=
[ True, False
, False, True, False, True
, True, False, False, True
, True, False, False
, True, False, True, False
] ^
([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**

⟨ML⟩

**definition** *test-wint32'* :: *wint32*  
**where**  $\langle \text{test-wint32}' = \text{drop-bit } 2 \ (\text{push-bit } 3 \ (0 + 10 - 14 * 3 \ \text{div } 6 \ \text{mod } 3)) \rangle$

⟨ML⟩

**lemma**  $x \ \text{AND} \ y = x \ \text{OR} \ (y :: \text{wint32})$   
**quickcheck** [*random, expect=counterexample*]  
**quickcheck** [*exhaustive, expect=counterexample*]  
 ⟨*proof*⟩

**lemma**  $(x :: \text{wint32}) \ \text{AND} \ x = x \ \text{OR} \ x$   
**quickcheck** [*narrowing, expect=no-counterexample*]  
 ⟨*proof*⟩

**lemma**  $(f :: \text{wint32} \Rightarrow \text{unit}) = g$   
**quickcheck** [*narrowing, size=3, expect=no-counterexample*]  
 ⟨*proof*⟩

**end**

## 9.5 Tests for *wint64*

**context**  
**includes** *bit-operations-syntax*  
**begin**

**definition** *test-wint64* :: *bool*  
**where**  $\langle \text{test-wint64} \longleftrightarrow$   
 (([ *0x10000000000000001*,  $-1$ ,  $-9223372036854775808$ , *0xFFFFFFFFFFFFFFFF*,  
*0x1234567890ABCDEF*  
 ,  $0x5A \ \text{AND} \ 0x36$   
 ,  $0x5A \ \text{OR} \ 0x36$   
 ,  $0x5A \ \text{XOR} \ 0x36$   
 ,  $\text{NOT } 0x5A$   
 ,  $5 + 6$ ,  $-5 + 6$ ,  $-6 + 5$ ,  $-5 + (-6)$ ,  $0xFFFFFFFFFFFFFFFF + 1$   
 ,  $5 - 3$ ,  $3 - 5$   
 ,  $5 * 3$ ,  $-5 * 3$ ,  $-5 * -4$ ,  $0x1234567890ABCDEF * 0xFEDCBA0987654321$   
 ,  $5 \ \text{div} \ 3$ ,  $-5 \ \text{div} \ 3$ ,  $-5 \ \text{div} \ -3$ ,  $5 \ \text{div} \ -3$   
 ,  $5 \ \text{mod} \ 3$ ,  $-5 \ \text{mod} \ 3$ ,  $-5 \ \text{mod} \ -3$ ,  $5 \ \text{mod} \ -3$   
 , *Bit-Operations.set-bit* 4 5, *Bit-Operations.set-bit* 2 (- 5)  
 , *Bit-Operations.set-bit* 32 5, *Bit-Operations.set-bit* 32 (- 5)  
 , *Bit-Operations.unset-bit* 0 5, *Bit-Operations.unset-bit* 1 (- 5)  
 , *Bit-Operations.unset-bit* 32 5, *Bit-Operations.unset-bit* 32 (- 5)  
 , *Bit-Operations.flip-bit* 4 5, *Bit-Operations.flip-bit* 1 (- 5)  
 , *Bit-Operations.flip-bit* 32 5, *Bit-Operations.flip-bit* 32 (- 5)  
 , *push-bit* 2 1, *push-bit* 3 (- 1), *push-bit* 64 1, *push-bit* 0 1  
 , *drop-bit* 3 100, *drop-bit* 3 (- 100), *drop-bit* 64 100, *drop-bit* 64 (- 100)

```

, signed-drop-bit-uint64 3 100, signed-drop-bit-uint64 3 (- 100)
, signed-drop-bit-uint64 64 100, signed-drop-bit-uint64 64 (- 100)
, take-bit 4 100, take-bit 4 (- 100)] :: 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, 4294967301, 18446744073709551611, 4, 18446744073709551609,
5, 18446744069414584315, 21, 18446744073709551609, 4294967301, 18446744069414584315
, 4, 18446744073709551608, 0, 1
, 12, 2305843009213693939, 0, 0
, 12, 18446744073709551603, 0, 18446744073709551615
, 4, 12]) ^
([ (0x5 :: uint64) = 0x5, (0x5 :: uint64) = 0x6
, (0x5 :: uint64) < 0x5, (0x5 :: uint64) < 0x6, (-5 :: uint64) < 6, (6 :: uint64)
< -5
, (0x5 :: uint64) ≤ 0x5, (0x5 :: uint64) ≤ 0x4, (-5 :: uint64) ≤ 6, (6 :: uint64)
≤ -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 0x7FFFFFFFFFFFFFFFFF, integer-of-uint64
0x8000000000000000, integer-of-uint64 0xAAAAAAAAAAAAAAAAAA])
=
[0, 0x7FFFFFFFFFFFFFFFFF, 0x8000000000000000, 0xAAAAAAAAAAAAAAAAAA])
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
, push-bit 2 1, push-bit 3 (- 1), push-bit 64 1, push-bit 0 1
, drop-bit 3 100, drop-bit 3 (- 100), drop-bit 64 100, drop-bit 64 (- 100)
, signed-drop-bit-uint64 3 100, signed-drop-bit-uint64 3 (- 100)
, signed-drop-bit-uint64 64 100, signed-drop-bit-uint64 64 (- 100)
, take-bit 4 100, take-bit 4 (- 100)] :: uint64 list

```

**export-code** *test-uint64* **checking** *SML Haskell? OCaml? Scala*

**notepad begin**

⟨*proof*⟩

**end**

⟨*ML*⟩

**definition** *test-uint64'* :: *uint64*

**where** ⟨*test-uint64' = drop-bit 2 (push-bit 3 (0 + 10 - 14 \* 3 div 6 mod 3))*⟩

⟨*ML*⟩

**end**

## 9.6 Tests for *wint*

**context**

**includes** *bit-operations-syntax*

**begin**

**definition** *test-wint* :: *bool*

**where** ⟨*test-wint = (let*

*test-list1 = (let*

*HS = wint-of-int (2 ^ (dflt-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 *dflt-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*

```

, Bit-Operations.set-bit dflt-size 5, Bit-Operations.set-bit dflt-size (- 5)
, Bit-Operations.unset-bit dflt-size 5, Bit-Operations.unset-bit dflt-size (- 5)
, Bit-Operations.flip-bit 0 5, Bit-Operations.flip-bit 0 (- 5)
, push-bit 2 1, push-bit 3 (- 1), push-bit dflt-size 1, push-bit 0 1
, drop-bit 3 31, drop-bit 3 (- 1), drop-bit dflt-size 31, drop-bit dflt-size (- 1)
, signed-drop-bit-uint 2 15, signed-drop-bit-uint 3 (- 1)
, signed-drop-bit-uint dflt-size 15, signed-drop-bit-uint dflt-size (- 1)
, take-bit 4 100, take-bit 4 (- 100)]
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 dflt-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)
    , 5, -5, 5, -5, 4, -6
    , 4, -8, 0, 1
    , 3, drop-bit 3 S - 1, 0, 0
    , 3, drop-bit 1 S + drop-bit 1 S - 1, 0, -1
    , 4, 12]
  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⟩

```

## 9.7 Tests for casts

```

definition test-casts :: bool
  where ⟨test-casts ↔
    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' ↔
    map uint8-of-uint16 [10, 0, 0xFE, 0xFFFF] = [10, 0, 0xFE, 0xFF] ∧
    map uint16-of-uint8 [10, 0, 0xFF] = [10, 0, 0xFF] ∧
    map uint16-of-uint32 [10, 0, 0xFFFE, 0xFFFFFFFF] = [10, 0, 0xFFFE, 0xFFFF]
  ∧
    map uint16-of-uint64 [10, 0, 0xFFFE, 0xFFFFFFFFFFFFFFFF] = [10, 0,
0xFFFE, 0xFFFF] ∧
    map uint32-of-uint16 [10, 0, 0xFFFF] = [10, 0, 0xFFFF] ∧
    map uint64-of-uint16 [10, 0, 0xFFFF] = [10, 0, 0xFFFF]⟩

```

```

definition test-casts'' :: bool
  where ⟨test-casts'' ↔
    map uint32-of-uint64 [10, 0, 0xFFFFFFFFFE, 0xFFFFFFFFFFFFFFFF] = [10,
0, 0xFFFFFFFFFE, 0xFFFFFFFF] ∧
    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 ↔
    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]) ∧
    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] ∧
    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] ∧
    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' ↔
    map uint-of-uint16 [0, 10, 0xFFFF] = [0, 10, 0xFFFF] ∧
    map uint16-of-uint [0, 10, 0xFFFF] = [0, 10, 0xFFFF]⟩

```

```

definition test-casts-uint'' :: bool

```

```

  where ⟨test-casts-uint'' ↔
    map uint-of-uint8 [0, 10, 0xFF] = [0, 10, 0xFF] ∧
    map uint8-of-uint [0, 10, 0xFF] = [0, 10, 0xFF]⟩

```

```

export-code test-casts-uint test-casts-uint'' checking SML Haskell?

```

```

export-code test-casts-uint checking OCaml?

```

```

export-code test-casts-uint' checking Haskell? Scala

```

```

end

```

```

end

```

```

end

```

```

theory Native-Word-Test-Emu

```

```

  imports

```

```

Native-Word-Test
begin

```

## 9.8 Test cases for emulation of native words

### 9.8.1 Tests for *uint8*

Test that *uint8* is emulated for OCaml via 8 word.

```

definition test-uint8-emulation :: bool
  where ⟨test-uint8-emulation ↔ (0xFFF - 0x10 = (0xEF :: uint8))⟩

```

```

export-code test-uint8-emulation checking OCaml?
  — test the other target languages as well SML Haskell? Scala

```

### 9.8.2 Tests for *uint16*

Test that *uint16* is emulated for PolyML and OCaml via 16 word.

```

definition test-uint16-emulation :: bool
  where ⟨test-uint16-emulation ↔ (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⟩

```

```

end

```

```

theory Native-Word-Test-PolyML
imports
  Native-Word-Test
begin

```

## 9.9 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

```

## 9.10 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 ↦ (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-GHC
imports
  Native-Word-Test
begin

```

## 9.11 Test with GHC

```

declare [[code-test-ghc = -XTypeSynonymInstances]]

```

```

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 GHC

```

### 9.11.1 Test quickcheck narrowing

```

context
  includes bit-operations-syntax
begin

```

```

lemma <(x :: uint64) AND y = x OR y>
quickcheck [narrowing, expect=counterexample]
<proof>

```

```

lemma <(f :: uint64 => bool) = g>
quickcheck [narrowing, size=3, expect=counterexample]
<proof>

```

```

lemma <(x :: uint32) AND y = x OR y>
quickcheck [narrowing, expect=counterexample]
<proof>

```

```

lemma ⟨(f :: uint32 ⇒ bool) = g⟩
quickcheck [narrowing, size=3, expect=counterexample]
  ⟨proof⟩

```

```

lemma ⟨(x :: uint16) AND y = x OR y⟩
quickcheck [narrowing, expect=counterexample]
  ⟨proof⟩

```

```

lemma ⟨(f :: uint16 ⇒ bool) = g⟩
quickcheck [narrowing, size=3, expect=counterexample]
  ⟨proof⟩

```

```

lemma ⟨(x :: uint8) AND y = x OR y⟩
quickcheck [narrowing, expect=counterexample]
  ⟨proof⟩

```

```

lemma ⟨(f :: uint8 ⇒ bool) = g⟩
quickcheck [narrowing, size=3, expect=counterexample]
  ⟨proof⟩

```

```

lemma ⟨(x :: uint) AND y = x OR y⟩
quickcheck [narrowing, expect=counterexample]
  ⟨proof⟩

```

```

lemma ⟨(f :: uint ⇒ bool) = g⟩
quickcheck [narrowing, size=3, expect=counterexample]
  ⟨proof⟩

```

```

end

```

```

end

```

```

theory Native-Word-Test-MLton
imports
  Native-Word-Test
begin

```

## 9.12 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-wint''
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-wint64 ⟨test-wint64' = 0x12⟩
    test-wint32 ⟨test-wint32' = 0x12⟩
    test-wint16
    test-wint8 ⟨test-wint8' = 0x12⟩
    test-wint
    test-casts
    test-casts'
    test-casts''
    test-casts-wint
    test-casts-wint'
    test-casts-wint''
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-wint16 test-wint16-emulation
    test-casts'
    test-casts-wint'
in MLton

```

```

end

```

```

theory Native-Word-Test-OCaml imports
    Native-Word-Test
begin

```

### 9.13 Test with OCaml

```

test-code
    test-wint64 ⟨test-wint64' = 0x12⟩
    test-wint32 ⟨test-wint32' = 0x12⟩

```

```

    test-uint
    test-casts''
    test-casts-uint
in OCaml

```

```
end
```

```
theory Native-Word-Test-OCaml2
```

```
  imports
```

```
    Native-Word-Test-Emu
```

```
begin
```

```
test-code
```

```
  test-uint16 test-uint16-emulation
```

```
  test-uint8 ⟨test-uint8' = 0x12⟩ test-uint8-emulation
```

```
  test-casts test-casts'
```

```
  test-casts-uint' test-casts-uint''
```

```
in OCaml
```

```
end
```

```
theory Native-Word-Test-SMLNJ
```

```
imports
```

```
  Native-Word-Test
```

```
begin
```

## 9.14 Test with SML/NJ

```
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 SMLNJ
```

SMLNJ provides a `Word64` structure. To test it in the `SML_word` target, we have to associate a driver with the combination. As SMLNJ does not implement a `Word16` structure, we remove the code module that refers to it. After this, `uint16` no longer works in the target `SML_word` as intended!

```
⟨ML⟩
```

```
code-printing code-module Uint16 → (SML-word) ⟨⟩
```

**test-code***test-uint64 test-uint64' = 0x12**test-uint32 test-uint32' = 0x12**test-uint8 test-uint8' = 0x12**test-uint*

— The cast operations for *uint64* use `Word64.fromLarge` and `Word64.toLarge`, which are unimplemented in SMLNJ's `Word64` structure.

*test-casts-uint''***in** *SMLNJ-word***end****theory** *Native-Word-Test-SMLNJ2***imports***Native-Word-Test-Emu***begin****test-code***test-uint16 test-uint16-emulation**test-casts'**test-casts-uint'***in** *SMLNJ***end**

## Chapter 10

# User guide for native words

This tutorial explains how to best use the types for native words like *uint32* 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 *uint32* instead of *32 word* and similarly for *uint64*, *uint16*, and *uint8*. 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, 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.

## 10.1 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

```

## 10.2 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.

### 10.2.1 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 e1 of Some (Bool b) ⇒ if b then eval e2 else eval e3
   | - ⇒ 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* *<proof>*

### 10.2.2 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.<sup>1</sup> 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.<sup>2</sup> 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* *<proof>*

**code-datatype** *Bool* *Word'*

**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*]

<sup>1</sup>Note that we should not declare a case translation for the new case combinator because this will break parsing case expressions with old case combinator.

<sup>2</sup>Case 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*  $\equiv dt\text{-case}\ c_1\ c_2\ \dots\ c_n$

**shows**  $(CASE\ (C_1\ a_{11}\ a_{12}\ \dots\ a_{1k_1}) \equiv c_1\ a_{11}\ a_{12}\ \dots\ a_{1k_1})$

$\&\&\&\ (CASE\ (C_2\ a_{21}\ a_{22}\ \dots\ a_{2k_2}) \equiv c_2\ a_{21}\ a_{22}\ \dots\ a_{2k_2})$

$\&\&\&\ \dots$

$\&\&\&\ (CASE\ (C_n\ a_{n1}\ a_{n2}\ \dots\ a_{nk_n}) \equiv c_n\ a_{n1}\ a_{n2}\ \dots\ a_{nk_n})$

```

lemma case-val'-cert:
  fixes bool word' b w
  assumes  $CASE \equiv case\text{-}val' \text{ bool } word'$ 
  shows  $(CASE (Bool\ b) \equiv bool\ b) \ \&\&\& \ (CASE (Word'\ w) \equiv word'\ w)$ 
   $\langle proof \rangle$ 

```

$\langle ML \rangle$

```

declare [[code drop: case-val]]

```

### 10.2.3 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

```

## 10.3 Troubleshooting

This section explains some possible problems when using native words. If you experience other difficulties, please contact the author.

### 10.3.1 *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  $\lambda x. x + x$   $\langle proof \rangle$ 

```

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>*

### 10.3.2 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 §10.3.1. For Haskell, you have to enable the extension `TypeSynonymInstances` with `-XTypeSynonymInstances` if you are using polymorphic bit operations on the native word types.

### 10.3.3 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.