

Native Words

Andreas Lochbihler

December 14, 2021

Abstract

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

Contents

1	A special case of a conversion.	5
2	Symbolic implementation of bit operations on int	7
2.1	Implementations of bit operations on <i>int</i> operating on symbolic representation	7
3	Bit operations for target language integers	11
3.1	More lemmas about <i>integers</i>	11
3.2	Bit operations on <i>integer</i>	12
3.3	Target language implementations	13
3.4	Test code generator setup	24
4	Common base for target language implementations of word types	27
5	Systematic approach towards type copies of word type	37
6	Unsigned words of 64 bits	45
6.1	Type definition and primitive operations	45
6.2	Code setup	48
6.3	Quickcheck setup	63
7	Unsigned words of 32 bits	65
7.1	Type definition and primitive operations	65
7.2	Code setup	68
7.3	Quickcheck setup	79
8	Unsigned words of 16 bits	81
8.1	Type definition and primitive operations	81
8.2	Code setup	84
8.3	Quickcheck setup	91
9	Unsigned words of 8 bits	93
9.1	Type definition and primitive operations	93

9.2	Code setup	96
9.3	Quickcheck setup	105
10	Unsigned words of default size	107
10.1	Type definition and primitive operations	108
10.2	Code setup	111
10.3	Quickcheck setup	123
11	Conversions between unsigned words and between char	125
11.1	Conversion between native words	126
11.2	Compatibility with Imperative/HOL	130
12	Test cases	131
12.1	Tests for <i>wint32</i>	131
12.2	Tests for <i>wint16</i>	133
12.3	Tests for <i>wint8</i>	135
12.4	Tests for <i>wint</i>	137
12.5	Tests for <i>wint64</i>	138
12.6	Tests for casts	141
13	Implementation of bit operations on int by target language operations	143
13.1	Test cases for emulation of native words	145
13.1.1	Tests for <i>wint16</i>	145
13.1.2	Tests for <i>wint8</i>	146
13.2	Test with PolyML	146
13.3	Test with Scala	147
14	User guide for native words	149
14.1	Lifting functions from <i>'a word</i> to native words	150
14.2	Storing native words in datatypes	151
14.2.1	Example: expressions and two semantics	151
14.2.2	Change the datatype to use machine words	152
14.2.3	Make functions use functions on machine words	153
14.3	Troubleshooting	153
14.3.1	<i>export-code</i> raises an exception	153
14.3.2	The generated code does not compile	154
14.3.3	The generated code is too slow	154

Chapter 1

A special case of a conversion.

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

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

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

```
lemma int-of-integer-symbolic-aux-code [code nbe]:
  int-of-integer-symbolic 0 = 0
  int-of-integer-symbolic (Code-Numeral.Pos n) = Int.Pos n
  int-of-integer-symbolic (Code-Numeral.Neg n) = Int.Neg n
by (simp-all add: int-of-integer-symbolic-def)
```

```
end
```


Chapter 2

Symbolic implementation of bit operations on int

```
theory Code-Symbolic-Bits-Int
imports
  Word-Lib.Least-significant-bit
  Word-Lib.Generic-set-bit
  Word-Lib.Bit-Comprehension
begin
```

2.1 Implementations of bit operations on *int* operating on symbolic representation

```
context
  includes bit-operations-syntax
begin
```

```
lemma test-bit-int-code [code]:
  bit (0::int)          n = False
  bit (Int.Neg num.One) n = True
  bit (Int.Pos num.One) 0 = True
  bit (Int.Pos (num.Bit0 m)) 0 = False
  bit (Int.Pos (num.Bit1 m)) 0 = True
  bit (Int.Neg (num.Bit0 m)) 0 = False
  bit (Int.Neg (num.Bit1 m)) 0 = True
  bit (Int.Pos num.One) (Suc n) = False
  bit (Int.Pos (num.Bit0 m)) (Suc n) = bit (Int.Pos m) n
  bit (Int.Pos (num.Bit1 m)) (Suc n) = bit (Int.Pos m) n
  bit (Int.Neg (num.Bit0 m)) (Suc n) = bit (Int.Neg m) n
  bit (Int.Neg (num.Bit1 m)) (Suc n) = bit (Int.Neg (Num.inc m)) n
  by (simp-all add: Num.add-One bit-Suc)
```

```
lemma int-not-code [code]:
  NOT (0 :: int) = -1
```

8CHAPTER 2. SYMBOLIC IMPLEMENTATION OF BIT OPERATIONS ON INT

$NOT (Int.Pos\ n) = Int.Neg (Num.inc\ n)$
 $NOT (Int.Neg\ n) = Num.sub\ n\ num.One$
by (*simp-all add: Num.add-One not-int-def*)

lemma *int-and-code* [*code*]: **fixes** $i\ j :: int$ **shows**

$0\ AND\ j = 0$
 $i\ AND\ 0 = 0$
 $Int.Pos\ n\ AND\ Int.Pos\ m = (case\ and-num\ n\ m\ of\ None\ \Rightarrow\ 0\ |\ Some\ n'\ \Rightarrow\ Int.Pos\ n')$
 $Int.Neg\ n\ AND\ Int.Neg\ m = NOT (Num.sub\ n\ num.One\ OR\ Num.sub\ m\ num.One)$
 $Int.Pos\ n\ AND\ Int.Neg\ num.One = Int.Pos\ n$
 $Int.Pos\ n\ AND\ Int.Neg (num.Bit0\ m) = Num.sub (or-not-num-neg (Num.BitM\ m)\ n)\ num.One$
 $Int.Pos\ n\ AND\ Int.Neg (num.Bit1\ m) = Num.sub (or-not-num-neg (num.Bit0\ m)\ n)\ num.One$
 $Int.Neg\ num.One\ AND\ Int.Pos\ m = Int.Pos\ m$
 $Int.Neg (num.Bit0\ n)\ AND\ Int.Pos\ m = Num.sub (or-not-num-neg (Num.BitM\ n)\ m)\ num.One$
 $Int.Neg (num.Bit1\ n)\ AND\ Int.Pos\ m = Num.sub (or-not-num-neg (num.Bit0\ n)\ m)\ num.One$
apply (*simp-all add: and-num-eq-None-iff [where ?'a = int] and-num-eq-Some-iff [where ?'a = int]*)
split: option.split
apply (*simp-all only: sub-one-eq-not-neg numeral-or-not-num-eq minus-minus and-not-numerals*)
bit.de-Morgan-disj bit.double-compl and-not-num-eq-None-iff and-not-num-eq-Some-iff ac-simps
apply *auto*
done

lemma *int-or-code* [*code*]: **fixes** $i\ j :: int$ **shows**

$0\ OR\ j = j$
 $i\ OR\ 0 = i$
 $Int.Pos\ n\ OR\ Int.Pos\ m = Int.Pos (or-num\ n\ m)$
 $Int.Neg\ n\ OR\ Int.Neg\ m = NOT (Num.sub\ n\ num.One\ AND\ Num.sub\ m\ num.One)$
 $Int.Pos\ n\ OR\ Int.Neg\ num.One = Int.Neg\ num.One$
 $Int.Pos\ n\ OR\ Int.Neg (num.Bit0\ m) = (case\ and-not-num (Num.BitM\ m)\ n\ of\ None\ \Rightarrow\ -1\ |\ Some\ n'\ \Rightarrow\ Int.Neg (Num.inc\ n'))$
 $Int.Pos\ n\ OR\ Int.Neg (num.Bit1\ m) = (case\ and-not-num (num.Bit0\ m)\ n\ of\ None\ \Rightarrow\ -1\ |\ Some\ n'\ \Rightarrow\ Int.Neg (Num.inc\ n'))$
 $Int.Neg\ num.One\ OR\ Int.Pos\ m = Int.Neg\ num.One$
 $Int.Neg (num.Bit0\ n)\ OR\ Int.Pos\ m = (case\ and-not-num (Num.BitM\ n)\ m\ of\ None\ \Rightarrow\ -1\ |\ Some\ n'\ \Rightarrow\ Int.Neg (Num.inc\ n'))$
 $Int.Neg (num.Bit1\ n)\ OR\ Int.Pos\ m = (case\ and-not-num (num.Bit0\ n)\ m\ of\ None\ \Rightarrow\ -1\ |\ Some\ n'\ \Rightarrow\ Int.Neg (Num.inc\ n'))$
apply (*auto simp add: numeral-or-num-eq split: option.splits*)
apply (*simp-all only: and-not-num-eq-None-iff and-not-num-eq-Some-iff and-not-numerals*)
numeral-or-not-num-eq or-int-def bit.double-compl ac-simps flip: nu-

2.1. IMPLEMENTATIONS OF BIT OPERATIONS ON INT OPERATING ON SYMBOLIC REPRESENTATION

```

meral-eq-iff [where ?'a = int])
  apply simp-all
done

```

```

lemma int-xor-code [code]: fixes i j :: int shows
  0 XOR j = j
  i XOR 0 = i
  Int.Pos n XOR Int.Pos m = (case xor-num n m of None => 0 | Some n' =>
Int.Pos n')
  Int.Neg n XOR Int.Neg m = Num.sub n num.One XOR Num.sub m num.One
  Int.Neg n XOR Int.Pos m = NOT (Num.sub n num.One XOR Int.Pos m)
  Int.Pos n XOR Int.Neg m = NOT (Int.Pos n XOR Num.sub m num.One)
  by (simp-all add: xor-num-eq-None-iff [where ?'a = int] xor-num-eq-Some-iff
[where ?'a = int] split: option.split)

```

```

lemma bin-rest-code: i div 2 = drop-bit 1 i for i :: int
  by (simp add: drop-bit-eq-div)

```

```

lemma set-bits-code [code]:
  set-bits = Code.abort (STR "set-bits is unsupported on type int") (λ-. set-bits ::
- => int)
  by simp

```

```

lemma fixes i :: int
  shows int-set-bit-True-conv-OR [code]: Generic-set-bit.set-bit i n True = i OR
push-bit n 1
  and int-set-bit-False-conv-NAND [code]: Generic-set-bit.set-bit i n False = i AND
NOT (push-bit n 1)
  and int-set-bit-conv-ops: Generic-set-bit.set-bit i n b = (if b then i OR (push-bit
n 1) else i AND NOT (push-bit n 1))
  by (simp-all add: bit-eq-iff) (auto simp add: bit-simps)

```

```

declare [[code drop: <drop-bit :: nat => int => int>]]

```

```

lemma drop-bit-int-code [code]: fixes i :: int shows
  drop-bit 0 i = i
  drop-bit (Suc n) 0 = (0 :: int)
  drop-bit (Suc n) (Int.Pos num.One) = 0
  drop-bit (Suc n) (Int.Pos (num.Bit0 m)) = drop-bit n (Int.Pos m)
  drop-bit (Suc n) (Int.Pos (num.Bit1 m)) = drop-bit n (Int.Pos m)
  drop-bit (Suc n) (Int.Neg num.One) = - 1
  drop-bit (Suc n) (Int.Neg (num.Bit0 m)) = drop-bit n (Int.Neg m)
  drop-bit (Suc n) (Int.Neg (num.Bit1 m)) = drop-bit n (Int.Neg (Num.inc m))
  by (simp-all add: drop-bit-Suc add-One)

```

```

declare [[code drop: <push-bit :: nat => int => int>]]

```

```

lemma push-bit-int-code [code]:
  push-bit 0 i = i

```

push-bit (*Suc n*) *i* = *push-bit n* (*Int.dup i*)
by (*simp-all add: ac-simps*)

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

lsb (*0 :: int*) = *False*
lsb (*Int.Pos num.One*) = *True*
lsb (*Int.Pos (num.Bit0 w)*) = *False*
lsb (*Int.Pos (num.Bit1 w)*) = *True*
lsb (*Int.Neg num.One*) = *True*
lsb (*Int.Neg (num.Bit0 w)*) = *False*
lsb (*Int.Neg (num.Bit1 w)*) = *True*
by *simp-all*

end

end

Chapter 3

Bit operations for target language integers

```
theory Bits-Integer imports
  Word-Lib.Bit-Comprehension
  Code-Int-Integer-Conversion
  Code-Symbolic-Bits-Int
begin

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
using assms unfolding Quotient-alt-def by blast

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

3.1 More lemmas about *integers*

```
context
includes integer.lifting
begin

lemma bitval-integer-transfer [transfer-rule]:
  (rel-fun (=) pcr-integer) of-bool of-bool
by(auto simp add: of-bool-def integer.pcr-cr-eq cr-integer-def)

lemma integer-of-nat-less-0-conv [simp]:  $\neg$  integer-of-nat n < 0
by(transfer) simp
```

lemma *int-of-integer-pow*: $\text{int-of-integer } (x \wedge n) = \text{int-of-integer } x \wedge n$
by(*induct n simp-all*)

lemma *pow-integer-transfer* [*transfer-rule*]:
 $(\text{rel-fun pcr-integer } (\text{rel-fun } (=) \text{ pcr-integer})) (\wedge) (\wedge)$
by(*auto 4 3 simp add: integer.pcr-cr-eq cr-integer-def int-of-integer-pow*)

lemma *sub1-lt-0-iff* [*simp*]: $\text{Code-Numeral.sub } n \text{ num.One} < 0 \longleftrightarrow \text{False}$
by(*cases n*)(*simp-all add: Code-Numeral.sub-code*)

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

lemma *nat-of-integer-sub1-conv-pred-numeral* [*simp*]:
 $\text{nat-of-integer } (\text{Code-Numeral.sub } n \text{ num.One}) = \text{pred-numeral } n$
by(*cases n*)(*simp-all add: Code-Numeral.sub-code*)

lemma *nat-of-integer-1* [*simp*]: $\text{nat-of-integer } 1 = 1$
by *transfer simp*

lemma *dup-1* [*simp*]: $\text{Code-Numeral.dup } 1 = 2$
by *transfer simp*

3.2 Bit operations on *integer*

Bit operations on *integer* are the same as on *int*

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

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

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

.

end

instantiation *integer* :: *lsb* **begin**
context includes *integer.lifting* **begin**

lift-definition *lsb-integer* :: $\text{integer} \Rightarrow \text{bool}$ **is** *lsb* .

instance
by (*standard; transfer*) (*fact lsb-odd*)

end

end

instantiation *integer* :: *msb* **begin**
context includes *integer.lifting* **begin**

lift-definition *msb-integer* :: $\text{integer} \Rightarrow \text{bool}$ **is** *msb* .

```

instance ..

end
end

instantiation integer :: set-bit begin
context includes integer.lifting begin

lift-definition set-bit-integer :: integer  $\Rightarrow$  nat  $\Rightarrow$  bool  $\Rightarrow$  integer is set-bit .

instance
  apply standard
  apply transfer
  apply (simp add: bit-simps)
  done

end
end

abbreviation (input) wf-set-bits-integer
where wf-set-bits-integer  $\equiv$  wf-set-bits-int

```

3.3 Target language implementations

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

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

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

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

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

```

code-identifier code-module Bits-Integer  $\rightarrow$ 
  (SML) Bits-Int and (OCaml) Bits-Int and (Haskell) Bits-Int and (Scala) Bits-Int

code-printing code-module Bits-Integer  $\rightarrow$  (SML)
 $\langle$ structure Bits-Integer : sig
  val set-bit : IntInf.int  $\rightarrow$  IntInf.int  $\rightarrow$  bool  $\rightarrow$  IntInf.int

```

14 CHAPTER 3. BIT OPERATIONS FOR TARGET LANGUAGE INTEGERS

```

    val shiftl : IntInf.int -> IntInf.int -> IntInf.int
    val shiftr : IntInf.int -> IntInf.int -> IntInf.int
    val test-bit : IntInf.int -> IntInf.int -> bool
end = struct

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

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

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

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

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

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

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

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

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

let test-bit x n = Z.testbit x (Z.to-int n);

end; (*struct Bits-Integer*)
code-reserved OCaml Bits-Integer

```

```

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

import qualified Data.Bits;

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

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

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

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

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

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

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

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

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

setBitBounded :: Data.Bits.Bits a => a -> Integer -> Bool -> a;
setBitBounded x n True = Data.Bits.setBit x (fromInteger n);
setBitBounded x n False = Data.Bits.clearBit x (fromInteger n);

```

```

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

shiftlBounded :: Data.Bits.Bits a => a -> Integer -> a;
shiftlBounded 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;

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

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

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

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

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

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

```

```
}>
```

code-reserved *Haskell Data-Bits*

code-printing code-module *Bits-Integer* \rightarrow (*Scala*)

```
{object Bits-Integer {
```

```

def setBit(x: BigInt, n: BigInt, b: Boolean) : BigInt =
  if (n.isValidInt)
    if (b)
      x.setBit(n.toInt)
    else
      x.clearBit(n.toInt)
  else
    sys.error("Bit index too large: " + n.toString)

```

```

def shiftrl(x: BigInt, n: BigInt) : BigInt =
  if (n.isValidInt)
    x << n.toInt
  else

```

18 CHAPTER 3. BIT OPERATIONS FOR TARGET LANGUAGE INTEGERS

```

    sys.error(Shift index too large: + n.toString)

def shiftr(x: BigInt, n: BigInt) : BigInt =
  if (n.isValidInt)
    x << n.toInt
  else
    sys.error(Shift index too large: + n.toString)

def testBit(x: BigInt, n: BigInt) : Boolean =
  if (n.isValidInt)
    x.testBit(n.toInt)
  else
    sys.error(Bit index too large: + n.toString)

} /* object Bits-Integer */

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

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

```

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

context

includes *integer.lifting bit-operations-syntax*

begin

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

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

$NOT\ i = -\ i - 1$

by *transfer (simp add: not-int-def)*

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

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

by *transfer rule*

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

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

by *transfer (simp add: and-one-eq odd-iff-mod-2-eq-one)*

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

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

by *transfer (simp add: odd-iff-mod-2-eq-one)*

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

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

by *transfer (simp add: and-one-eq mod-2-eq-odd)*

end

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

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

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

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

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

by (*simp add: integer-test-bit-def*)

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

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

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

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

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

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

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

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

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

```

integer-test-bit (Code-Numeral.Pos n) (Code-Numeral.sub n' num.One)
integer-test-bit (Code-Numeral.Pos (num.Bit1 n)) (Code-Numeral.Pos n') =
integer-test-bit (Code-Numeral.Pos n) (Code-Numeral.sub n' num.One)
integer-test-bit (Code-Numeral.Neg num.One) 0 = True
integer-test-bit (Code-Numeral.Neg (num.Bit0 n)) 0 = False
integer-test-bit (Code-Numeral.Neg (num.Bit1 n)) 0 = True
integer-test-bit (Code-Numeral.Neg num.One) (Code-Numeral.Pos n') = True
integer-test-bit (Code-Numeral.Neg (num.Bit0 n)) (Code-Numeral.Pos n') =
integer-test-bit (Code-Numeral.Neg n) (Code-Numeral.sub n' num.One)
integer-test-bit (Code-Numeral.Neg (num.Bit1 n)) (Code-Numeral.Pos n') =
integer-test-bit (Code-Numeral.Neg (n + num.One)) (Code-Numeral.sub n' num.One)
by (simp-all add: integer-test-bit-def bit-integer-def flip: bit-not-int-iff')

```

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

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

lemma *lsb-integer-code* [code]:
fixes *x* :: *integer* **shows**
 $lsb\ x = bit\ x\ 0$
by *transfer(simp add: lsb-int-def)*

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

lemma *set-bit-integer-code* [code]:
 $set-bit\ x\ i\ b = integer-set-bit\ x\ (integer-of-nat\ i)\ b$
by(*simp add: integer-set-bit-def*)

lemma *set-bit-integer-conv-masks*:
fixes *x* :: *integer* **shows**
 $set-bit\ x\ i\ b = (if\ b\ then\ x\ OR\ (push-bit\ i\ 1)\ else\ x\ AND\ NOT\ (push-bit\ i\ 1))$
by *transfer (simp add: int-set-bit-False-conv-NAND int-set-bit-True-conv-OR)*

end

code-printing constant *integer-set-bit* \rightarrow
(SML) *Bits'-Integer.set'-bit* **and**
(Haskell) (*Data'-Bits.setBitUnbounded* :: *Integer* \rightarrow *Integer* \rightarrow *Bool* \rightarrow *Integer*) **and**

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

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

context

includes *bit-operations-syntax*

begin

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

```
integer-set-bit x n b =
  (if n < 0 then undefined x n b else
   if b then x OR (push-bit (nat-of-integer n) 1)
   else x AND NOT (push-bit (nat-of-integer n) 1))
  by (auto simp add: integer-set-bit-def not-less set-bit-eq set-bit-def unset-bit-def)
```

end

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

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

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

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

```
fixes x :: integer shows
  push-bit n x = integer-shiffl x (integer-of-nat n)
```

by(auto simp add: integer-shiffl-def)

context

includes *integer.lifting*

begin

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

```
fixes x :: integer shows
  push-bit n x = x * 2 ^ n
  by (fact push-bit-eq-mult)
```

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

```
integer-shiffl x (Code-N numeral.Neg n) = undefined x (Code-N numeral.Neg n)
integer-shiffl x 0 = x
integer-shiffl x (Code-N numeral.Pos n) = integer-shiffl (Code-N numeral.dup x)
(Code-N numeral.sub n num.One)
integer-shiffl 0 (Code-N numeral.Pos n) = 0
  apply (simp-all add: integer-shiffl-def numeral-eq-Suc)
  apply transfer
  apply (simp add: ac-simps)
```

done

end

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

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

declare [[*code drop*: \langle drop-bit :: nat \Rightarrow integer \Rightarrow integer \rangle]]

lemma *shiftr-integer-conv-div-pow2*:
includes *integer.lifting* **fixes** *x* :: *integer* **shows**
drop-bit n x = *x div 2 ^ n*
by (*fact drop-bit-eq-div*)

lemma *shiftr-integer-code* [*code*]:
fixes *x* :: *integer* **shows**
drop-bit n x = *integer-shiftr x (integer-of-nat n)*
by(*auto simp add: integer-shiftr-def*)

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

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

```

integer-shiftr (Code-Numeral.Neg n') (Code-Numeral.sub n num.One)
integer-shiftr (Code-Numeral.Neg (num.Bit1 n')) (Code-Numeral.Pos n) =
integer-shiftr (Code-Numeral.Neg (Num.inc n')) (Code-Numeral.sub n num.One)
  apply (simp-all add: integer-shiftr-def numeral-eq-Suc drop-bit-Suc)
  apply transfer apply simp
  apply transfer apply simp
  apply transfer apply (simp add: add-One)
done

```

```

context
includes integer.lifting
begin

```

```

lemma Bit-integer-code [code]:
  Bit-integer i False = push-bit 1 i
  Bit-integer i True = push-bit 1 i + 1
  by (transfer; simp)+

```

```

lemma msb-integer-code [code]:
  msb (x :: integer)  $\longleftrightarrow$  x < 0
  by transfer (simp add: msb-int-def)

```

```
end
```

```

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

```

```

lemma bitAND-integer-unfold [code]:
  x AND y =
    (if x = 0 then 0
     else if x = - 1 then y
     else Bit-integer (bin-rest-integer x AND bin-rest-integer y) (bin-last-integer x  $\wedge$ 
bin-last-integer y))
  by transfer
    (auto simp add: algebra-simps
and-int-rec [of - <- * 2>] and-int-rec [of <- * 2>] and-int-rec [of <1 + - * 2>]
elim!: evenE oddE)

```

```

lemma bitOR-integer-unfold [code]:
  x OR y =
    (if x = 0 then y
     else if x = - 1 then - 1
     else Bit-integer (bin-rest-integer x OR bin-rest-integer y) (bin-last-integer x  $\vee$ 
bin-last-integer y))
  by transfer
    (auto simp add: algebra-simps
or-int-rec [of - <- * 2>] or-int-rec [of - <1 + - * 2>] or-int-rec [of <1 + - * 2>]
elim!: evenE oddE)

```

```

lemma bitXOR-integer-unfold [code]:
  x XOR y =
    (if x = 0 then y
     else if x = - 1 then NOT y
     else Bit-integer (bin-rest-integer x XOR bin-rest-integer y)
      (¬ bin-last-integer x ↔ bin-last-integer y))
by transfer
  (auto simp add: algebra-simps
   xor-int-rec [of - <- * 2>] xor-int-rec [of <- * 2>] xor-int-rec [of <1 + - * 2>]
   elim!: evenE oddE)

end

```

3.4 Test code generator setup

```

context
  includes bit-operations-syntax
begin

```

```

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

```

```

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

```

```

notepad begin
have bit-integer-test by eval

```



```
have bit-integer-test by normalization
have bit-integer-test by code-simp
end
ML-val ⟨val true = @{code bit-integer-test}⟩

lemma x AND y = x OR (y :: integer)
quickcheck[random, expect=counterexample]
quickcheck[exhaustive, expect=counterexample]
oops

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

lemma (f :: integer ⇒ unit) = g
quickcheck[narrowing, size=3, expect=no-counterexample]
by(simp add: fun-eq-iff)

end

hide-const bit-integer-test
hide-fact bit-integer-test-def

end
```


Chapter 4

Common base for target language implementations of word types

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

More lemmas

```
lemma div-half-nat:  
  fixes  $x\ y :: \text{nat}$   
  assumes  $y \neq 0$   
  shows  $(x \text{ div } y, x \text{ mod } y) = (\text{let } q = 2 * (x \text{ div } 2 \text{ div } y); r = x - q * y \text{ in if } y \leq r \text{ then } (q + 1, r - y) \text{ else } (q, r))$   
proof -  
  let  $?q = 2 * (x \text{ div } 2 \text{ div } y)$   
  have  $q: ?q = x \text{ div } y - x \text{ div } y \text{ mod } 2$   
    by (metis div-mult2-eq mult.commute minus-mod-eq-mult-div [symmetric])  
  let  $?r = x - ?q * y$   
  have  $r: ?r = x \text{ mod } y + x \text{ div } y \text{ mod } 2 * y$   
    by (simp add: q diff-mult-distrib minus-mod-eq-div-mult [symmetric])(metis diff-diff-cancel mod-less-eq-dividend mod-mult2-eq add.commute mult.commute)  
  
  show ?thesis  
  proof (cases  $y \leq x - ?q * y$ )  
    case True  
    with assms  $q$  have  $x \text{ div } y \text{ mod } 2 \neq 0$  unfolding  $r$   
    by (metis Nat.add-0-right diff-0-eq-0 diff-Suc-1 le-div-geq mod2-gr-0 mod-div-trivial mult-0 neq0-conv numeral-1-eq-Suc-0 numerals(1))  
    hence  $x \text{ div } y = ?q + 1$  unfolding  $q$   
    by simp
```

```

moreover hence  $x \bmod y = ?r - y$ 
  by (simp (metis minus-div-mult-eq-mod [symmetric] diff-commute diff-diff-left
mult-Suc))
  ultimately show ?thesis using True by (simp add: Let-def)
next
  case False
  hence  $x \operatorname{div} y \bmod 2 = 0$  unfolding r
    by (simp add: not-le) (metis Nat.add-0-right assms div-less div-mult-self2
mod-div-trivial mult.commute)
    hence  $x \operatorname{div} y = ?q$  unfolding q by simp
    moreover hence  $x \bmod y = ?r$  by (metis minus-div-mult-eq-mod [symmetric])
    ultimately show ?thesis using False by (simp add: Let-def)
qed
qed

```

lemma *div-half-word*:

```

fixes  $x\ y :: 'a :: \text{len word}$ 
assumes  $y \neq 0$ 
shows  $(x \operatorname{div} y, x \bmod y) = (\text{let } q = \text{push-bit } 1 \ (\text{drop-bit } 1 \ x \operatorname{div} y); r = x - q$ 
 $* y \text{ in if } y \leq r \text{ then } (q + 1, r - y) \text{ else } (q, r))$ 
proof –
  obtain  $n$  where  $n: x = \text{of-nat } n \quad n < 2^{\wedge} \text{LENGTH}('a)$ 
    by (rule that [of unat x]) simp-all
  moreover obtain  $m$  where  $m: y = \text{of-nat } m \quad m < 2^{\wedge} \text{LENGTH}('a)$ 
    by (rule that [of unat y]) simp-all
  ultimately have [simp]:  $\langle \text{unat } (\text{of-nat } n :: 'a \text{ word}) = n \rangle \langle \text{unat } (\text{of-nat } m :: 'a$ 
 $\text{word}) = m \rangle$ 
    by (transfer, simp add: take-bit-of-nat take-bit-nat-eq-self-iff) +
  let  $?q = \text{push-bit } 1 \ (\text{drop-bit } 1 \ x \operatorname{div} y)$ 
  let  $?q' = 2 * (n \operatorname{div} 2 \operatorname{div} m)$ 
  have  $n \operatorname{div} 2 \operatorname{div} m < 2^{\wedge} \text{LENGTH}('a)$ 
    using  $n$  by (metis of-nat-inverse unat-lt2p uno-simps(2))
  hence  $q: ?q = \text{of-nat } ?q'$  using  $n\ m$ 
  by (auto simp add: drop-bit-eq-div word-arith-nat-div uno-simps take-bit-nat-eq-self
unsigned-of-nat)
  from assms have  $m \neq 0$  using  $m$  by –(rule notI, simp)

from  $n$  have  $2 * (n \operatorname{div} 2 \operatorname{div} m) < 2^{\wedge} \text{LENGTH}('a)$ 
  by (metis mult.commute div-mult2-eq minus-mod-eq-mult-div [symmetric] less-imp-diff-less
of-nat-inverse unat-lt2p uno-simps(2))
moreover
  have  $2 * (n \operatorname{div} 2 \operatorname{div} m) * m < 2^{\wedge} \text{LENGTH}('a)$  using  $n$  unfolding div-mult2-eq [symmetric]
    by (subst (2) mult.commute) (simp add: minus-mod-eq-div-mult [symmetric]
diff-mult-distrib minus-mod-eq-mult-div [symmetric] div-mult2-eq)
  moreover have  $2 * (n \operatorname{div} 2 \operatorname{div} m) * m \leq n$ 
    by (simp flip: div-mult2-eq ac-simps)
ultimately
have  $r: x - ?q * y = \text{of-nat } (n - ?q' * m)$ 
  and  $y \leq x - ?q * y \implies \text{of-nat } (n - ?q' * m) - y = \text{of-nat } (n - ?q' * m -$ 

```

m)
using *n m unfolding q*
apply (*simp-all add: of-nat-diff*)
apply (*subst of-nat-diff*)
apply (*simp-all add: word-le-nat-alt take-bit-nat-eq-self unat-sub-if' unat-word-ariths unsigned-of-nat*)
done
then show *?thesis using n m div-half-nat [OF ⟨m ≠ 0⟩, of n] unfolding q*
by (*simp add: word-le-nat-alt word-div-def word-mod-def Let-def take-bit-nat-eq-self unsigned-of-nat*
flip: zdiv-int zmod-int
split del: if-split split: if-split-asm)
qed

lemma *word-test-bit-set-bits: bit (BITS n. f n :: 'a :: len word) n \longleftrightarrow n < LENGTH('a) \wedge f n*
by (*fact bit-set-bits-word-iff*)

lemma *word-of-int-conv-set-bits: word-of-int i = (BITS n. bit i n)*
by (*rule word-eqI*) (*auto simp add: word-test-bit-set-bits bit-simps*)

context
includes *bit-operations-syntax*
begin

lemma *word-and-mask-or-conv-and-mask:*
bit n index \implies (n AND mask index) OR (push-bit index 1) = n AND mask (index + 1)
for *n :: ⟨'a::len word⟩*
by(*rule word-eqI*)(*auto simp add: bit-simps*)

lemma *uint-and-mask-or-full:*
fixes *n :: 'a :: len word*
assumes *bit n (LENGTH('a) - 1)*
and *mask1 = mask (LENGTH('a) - 1)*
and *mask2 = push-bit (LENGTH('a) - 1) 1*
shows *uint (n AND mask1) OR mask2 = uint n*
proof –
have *mask2 = uint (push-bit (LENGTH('a) - 1) 1 :: 'a word) using assms*
by *transfer (simp add: take-bit-push-bit)*
hence *uint (n AND mask1) OR mask2 = uint (n AND mask1 OR (push-bit (LENGTH('a) - 1) 1 :: 'a word))*
by(*simp add: uint-or*)
also have *... = uint (n AND mask (LENGTH('a) - 1 + 1))*
using *assms by (simp only: word-and-mask-or-conv-and-mask)*
also have *... = uint n by simp*
finally show *?thesis .*
qed

end

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

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

lemma [*code*]:

```

x sdiv y =
  (let x' = sint x; y' = sint y;
   negative = (x' < 0) ≠ (y' < 0);
   result = abs x' div abs y'
   in word-of-int (if negative then -result else result))
for x y :: ⟨'a::len word⟩
by (simp add: sdiv-word-def signed-divide-int-def sgn-if Let-def not-less not-le)

```

lemma [*code*]:

```

x smod y =
  (let x' = sint x; y' = sint y;
   negative = (x' < 0);
   result = abs x' mod abs y'
   in word-of-int (if negative then -result else result))
for x y :: ⟨'a::len word⟩
proof –
  have *: ⟨k mod l = k - k div l * l⟩ for k l :: int
  by (simp add: minus-div-mult-eq-mod)
  show ?thesis
  by (simp add: smod-word-def signed-modulo-int-def signed-divide-int-def * sgn-if Let-def)
qed

```

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

lemma *divmod-via-sdivmod*:

```

fixes x y :: 'a :: len word
assumes y ≠ 0
shows
  (x div y, x mod y =
   (if push-bit (LENGTH('a) - 1) 1 ≤ y then if x < y then (0, x) else (1, x - y)
    else let q = (push-bit 1 (drop-bit 1 x sdiv y));
     r = x - q * y
     in if r ≥ y then (q + 1, r - y) else (q, r))
  proof(cases push-bit (LENGTH('a) - 1) 1 ≤ y)
  case True
  note y = this
  show ?thesis
  proof(cases x < y)
  case True
  then have x mod y = x

```

```

    by transfer simp
  thus ?thesis using True y by (simp add: word-div-lt-eq-0)
next
case False
obtain n where n: y = of-nat n    n < 2 ^ LENGTH('a)
  by (rule that [of ⟨unat y⟩]) simp-all
have unat x < 2 ^ LENGTH('a) by (rule unat-lt2p)
also have ... = 2 * 2 ^ (LENGTH('a) - 1)
  by (metis Suc-pred len-gt-0 power-Suc One-nat-def)
also have ... ≤ 2 * n using y n
  by transfer (simp add: push-bit-of-1 take-bit-eq-mod)
finally have div: x div of-nat n = 1 using False n
  by (simp add: word-div-eq-1-iff take-bit-nat-eq-self unsigned-of-nat)
moreover have x mod y = x - x div y * y
  by (simp add: minus-div-mult-eq-mod)
with div n have x mod y = x - y by simp
ultimately show ?thesis using False y n by simp
qed
next
case False
note y = this
obtain n where n: x = of-nat n    n < 2 ^ LENGTH('a)
  by (rule that [of ⟨unat x⟩]) simp-all
hence int n div 2 + 2 ^ (LENGTH('a) - Suc 0) < 2 ^ LENGTH('a)
  by (cases ⟨LENGTH('a)⟩)
    (auto dest: less-imp-of-nat-less [where ?'a = int])
with y n have sint (drop-bit 1 x) = uint (drop-bit 1 x)
  by (cases ⟨LENGTH('a)⟩)
    (auto simp add: sint-uint drop-bit-eq-div take-bit-nat-eq-self uint-div-distrib
      signed-take-bit-int-eq-self-iff unsigned-of-nat)
moreover have uint y + 2 ^ (LENGTH('a) - Suc 0) < 2 ^ LENGTH('a)
  using y by (cases ⟨LENGTH('a)⟩)
    (simp-all add: not-le push-bit-of-1 word-less-alt uint-power-lower)
then have sint y = uint y
  apply (cases ⟨LENGTH('a)⟩)
  apply (auto simp add: sint-uint signed-take-bit-int-eq-self-iff)
  using uint-ge-0 [of y]
  by linarith
ultimately show ?thesis using y
  apply (subst div-half-word [OF assms])
  apply (simp add: sdw-word-def signed-divide-int-def flip: uint-div)
  done
qed

```

More implementations tailored towards target-language implementations

```

context
  includes integer.lifting
begin

```

lift-definition *word-of-integer* :: *integer* \Rightarrow '*a* :: *len word* **is** *word-of-int* .

lemma *word-of-integer-code* [*code*]: *word-of-integer* *n* = *word-of-int* (*int-of-integer* *n*)
by (*simp add: word-of-integer.rep-eq*)

end

context
includes *bit-operations-syntax*
begin

lemma *word-of-int-code*:
uint (*word-of-int* *x* :: '*a* *word*) = *x* **AND** *mask* (*LENGTH*('a :: *len*))
by (*simp add: unsigned-of-int take-bit-eq-mask*)

context
fixes *f* :: *nat* \Rightarrow *bool*
begin

definition *set-bits-aux* :: \langle *nat* \Rightarrow '*a* *word* \Rightarrow '*a*::*len word* \rangle
where \langle *set-bits-aux* *n* *w* = *push-bit* *n* *w* **OR** *take-bit* *n* (*set-bits* *f*) \rangle

lemma *bit-set-bit-aux* [*bit-simps*]:
 \langle *bit* (*set-bits-aux* *n* *w*) *m* \longleftrightarrow *m* < *LENGTH*('a) \wedge
(*if* *m* < *n* *then* *f* *m* *else* *bit* *w* (*m* - *n*)) \rangle **for** *w* :: '*a*::*len word* \rangle
by (*auto simp add: bit-simps set-bits-aux-def*)

corollary *set-bits-conv-set-bits-aux*:
 \langle *set-bits* *f* = (*set-bits-aux* *LENGTH*('a) 0 :: '*a* :: *len word*) \rangle
by (*rule bit-word-eqI*) (*simp add: bit-simps*)

lemma *set-bits-aux-0* [*simp*]:
 \langle *set-bits-aux* 0 *w* = *w* \rangle
by (*simp add: set-bits-aux-def*)

lemma *set-bits-aux-Suc* [*simp*]:
 \langle *set-bits-aux* (*Suc* *n*) *w* = *set-bits-aux* *n* (*push-bit* 1 *w* **OR** (*if* *f* *n* *then* 1 *else* 0)) \rangle
by (*rule bit-word-eqI*) (*auto simp add: bit-simps le-less-Suc-eq mult.commute* [*of*
- 2])

lemma *set-bits-aux-simps* [*code*]:
 \langle *set-bits-aux* 0 *w* = *w* \rangle
 \langle *set-bits-aux* (*Suc* *n*) *w* = *set-bits-aux* *n* (*push-bit* 1 *w* **OR** (*if* *f* *n* *then* 1 *else* 0)) \rangle
by *simp-all*

lemma *set-bits-aux-rec*:
 \langle *set-bits-aux* *n* *w* =
(*if* *n* = 0 *then* *w*

else let $n' = n - 1$ in set-bits-aux n' (push-bit 1 w OR (if $f n'$ then 1 else 0)))
 by (cases n) simp-all

end

lemma word-of-int-via-signed:

fixes $mask$
 assumes $mask\text{-def}$: $mask = \text{Bit-Operations.mask } (\text{LENGTH } 'a)$
 and $shift\text{-def}$: $shift = \text{push-bit } \text{LENGTH } 'a \ 1$
 and $index\text{-def}$: $index = \text{LENGTH } 'a - 1$
 and $overflow\text{-def}$: $overflow = \text{push-bit } (\text{LENGTH } 'a - 1) \ 1$
 and $least\text{-def}$: $least = - \ overflow$
 shows
 ($\text{word-of-int } i :: 'a :: \text{len word}$) =
 (let $i' = i \ \text{AND } mask$
 in if bit i' $index$ then
 if $i' - shift < least \vee overflow \leq i' - shift$ then arbitrary1 i' else word-of-int
 ($i' - shift$)
 else if $i' < least \vee overflow \leq i'$ then arbitrary2 i' else $\text{word-of-int } i'$)
 proof –
 define i' where $i' = i \ \text{AND } mask$
 have $shift = mask + 1$ unfolding $assms$
 by (simp add: $mask\text{-eq-exp-minus-1 push-bit-of-1}$)
 hence $i' < shift$
 by (simp add: $mask\text{-def } i'\text{-def}$)
 show ?thesis
 proof (cases bit i' $index$)
 case True
 then have unf : $i' = overflow \ \text{OR } i'$
 apply (simp add: $assms \ i'\text{-def push-bit-of-1 flip: take-bit-eq-mask}$)
 apply (rule bit-eqI)
 apply (auto simp add: $bit\text{-take-bit-iff bit-or-iff bit-exp-iff}$)
 done
 have $\langle overflow \leq overflow \ \text{OR } i' \rangle$
 by (simp add: $i'\text{-def } mask\text{-def or-greater-eq}$)
 then have $overflow \leq i'$
 by (subst unf)
 hence $i' - shift < least \longleftrightarrow False$ unfolding $assms$
 by (cases $\text{LENGTH } 'a$) (simp-all add: $not\text{-less push-bit-of-1}$)
 moreover
 have $overflow \leq i' - shift \longleftrightarrow False$ using $\langle i' < shift \rangle$ unfolding $assms$
 by (cases $\text{LENGTH } 'a$) (auto simp add: $not\text{-le push-bit-of-1 elim: less-le-trans}$)
 moreover
 have $\text{word-of-int } (i' - shift) = (\text{word-of-int } i :: 'a \ \text{word})$ using $\langle i' < shift \rangle$
 by (simp add: $i'\text{-def } shift\text{-def } mask\text{-def push-bit-of-1 word-of-int-eq-iff flip:$
 $take\text{-bit-eq-mask}$)
 ultimately show ?thesis using True by (simp add: $Let\text{-def } i'\text{-def}$)
 next
 case False

```

have  $i' = i$  AND Bit-Operations.mask (LENGTH('a) - 1)
  apply (rule bit-eqI)
  apply (use False in <auto simp add: bit-simps assms i'-def>)
  apply (auto simp add: less-le)
  done
also have  $\dots \leq$  Bit-Operations.mask (LENGTH('a) - 1)
  using AND-upper2 mask-nonnegative-int by blast
also have  $\dots <$  overflow
  by (simp add: mask-int-def push-bit-of-1 overflow-def)
also
have  $least \leq 0$  unfolding least-def overflow-def by simp
have  $0 \leq i'$  by (simp add: i'-def mask-def)
hence  $least \leq i'$  using <least ≤ 0> by simp
moreover
have word-of-int i' = (word-of-int i :: 'a word)
  by (simp add: i'-def mask-def of-int-and-eq of-int-mask-eq)
ultimately show ?thesis using False by(simp add: Let-def i'-def)
qed
qed

end

```

Quickcheck conversion functions

```

context
  includes state-combinator-syntax
begin

definition qc-random-cnv ::
  (natural  $\Rightarrow$  'a::term-of)  $\Rightarrow$  natural  $\Rightarrow$  Random.seed
   $\Rightarrow$  ('a  $\times$  (unit  $\Rightarrow$  Code-Evaluation.term))  $\times$  Random.seed
  where qc-random-cnv a-of-natural i = Random.range (i + 1)  $\circ$   $\rightarrow$  ( $\lambda k. \text{Pair} ($ 
    let n = a-of-natural k
    in (n,  $\lambda \cdot. \text{Code-Evaluation.term-of } n$ )))

end

definition qc-exhaustive-cnv :: (natural  $\Rightarrow$  'a)  $\Rightarrow$  ('a  $\Rightarrow$  (bool  $\times$  term list) option)
   $\Rightarrow$  natural  $\Rightarrow$  (bool  $\times$  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  $\Rightarrow$  ('a::term-of))  $\Rightarrow$  ('a  $\times$  (unit  $\Rightarrow$  term)  $\Rightarrow$  (bool  $\times$  term list) option)
   $\Rightarrow$  natural  $\Rightarrow$  (bool  $\times$  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])
by pat-completeness auto
termination including integer.lifting
proof(relation measure nat-of-integer)
  fix i :: integer
  assume 0 < i
  thus (i - 1, i) ∈ measure nat-of-integer
  by simp(transfer, simp)
qed simp

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)
by (rule partial-term-of-anything)+

end

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

```

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

```
setup <Code-Target.add-derived-target (SML-word, [(Code-ML.target-SML, I)])>
```

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

```
end
```

Chapter 5

Systematic approach towards type copies of word type

```
theory Word-Type-Copies
imports
  HOL-Library.Word
  Word-Lib.Most-significant-bit
  Word-Lib.Least-significant-bit
  Word-Lib.Generic-set-bit
  Code-Target-Word-Base
begin

lemma int-of-integer-unsigned-eq [simp]:
   $\langle \text{int-of-integer } (\text{unsigned } w) = \text{uint } w \rangle$ 
  by transfer simp

lemma int-of-integer-signed-eq [simp]:
   $\langle \text{int-of-integer } (\text{signed } w) = \text{sint } w \rangle$ 
  by transfer simp

lemma word-of-int-of-integer-eq [simp]:
   $\langle \text{word-of-int } (\text{int-of-integer } k) = \text{word-of-integer } k \rangle$ 
  by (simp add: word-of-integer.rep-eq)
```

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

```
locale word-type-copy =
  fixes of-word ::  $\langle 'b::\text{len } \text{word} \Rightarrow 'a \rangle$ 
  and word-of ::  $\langle 'a \Rightarrow 'b \text{ word} \rangle$ 
  assumes type-definition:  $\langle \text{type-definition } \text{word-of } \text{of-word } \text{UNIV} \rangle$ 
begin

lemma word-of-word:
   $\langle \text{word-of } (\text{of-word } w) = w \rangle$ 
  using type-definition by (simp add: type-definition-def)
```

```

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.
  using type-definition by (simp add: type-definition-def)

```

```

lemma word-of-eqI:
  ⟨p = q⟩ if ⟨word-of p = word-of q⟩
proof —
  from that have ⟨of-word (word-of p) = of-word (word-of q)⟩
  by simp
  then show ?thesis
  by (simp add: of-word-of)
qed

```

```

lemma eq-iff-word-of:
  ⟨p = q ↔ word-of p = word-of q⟩
  by (auto intro: word-of-eqI)

```

```

end

```

```

bundle constraintless
begin

```

```

declaration ⟨
  let
    val cs = map (rpair NONE o fst o dest-Const)
      [ term ⟨0⟩, term ⟨(+)⟩, term ⟨uminus⟩, term ⟨(-)⟩,
        term ⟨1⟩, term ⟨(*)⟩, term ⟨(div)⟩, term ⟨(mod)⟩,
        term ⟨HOL.equal⟩, term ⟨(≤)⟩, term ⟨(<)⟩,
        term ⟨(dvd)⟩, term ⟨of-bool⟩, term ⟨numeral⟩, term ⟨of-nat⟩,
        term ⟨bit⟩,
        term ⟨Bit-Operations.not⟩, term ⟨Bit-Operations.and⟩, term ⟨Bit-Operations.or⟩,
term ⟨Bit-Operations.xor⟩, term ⟨mask⟩,
        term ⟨push-bit⟩, term ⟨drop-bit⟩, term ⟨take-bit⟩,
        term ⟨Bit-Operations.set-bit⟩, term ⟨unset-bit⟩, term ⟨flip-bit⟩,
        term ⟨msb⟩, term ⟨lsb⟩, term ⟨size⟩, term ⟨Generic-set-bit.set-bit⟩, term ⟨set-bits⟩ ]
    in
      K (Context.mapping I (fold Proof-Context.add-const-constraint cs))
    end
  ⟩

```

```

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]:  $\langle \text{word-of } 1 = 1 \rangle$ 
and word-of-add [code]:  $\langle \text{word-of } (p + q) = \text{word-of } p + \text{word-of } q \rangle$ 
and word-of-minus [code]:  $\langle \text{word-of } (-p) = -(\text{word-of } p) \rangle$ 
and word-of-diff [code]:  $\langle \text{word-of } (p - q) = \text{word-of } p - \text{word-of } q \rangle$ 
and word-of-mult [code]:  $\langle \text{word-of } (p * q) = \text{word-of } p * \text{word-of } q \rangle$ 
and word-of-div [code]:  $\langle \text{word-of } (p \text{ div } q) = \text{word-of } p \text{ div } \text{word-of } q \rangle$ 
and word-of-mod [code]:  $\langle \text{word-of } (p \text{ mod } q) = \text{word-of } p \text{ mod } \text{word-of } q \rangle$ 
and equal-iff-word-of [code]:  $\langle \text{HOL.equal } p \ q \longleftrightarrow \text{HOL.equal } (\text{word-of } p)$ 
(word-of } q) \rangle
and less-eq-iff-word-of [code]:  $\langle p \leq q \longleftrightarrow \text{word-of } p \leq \text{word-of } q \rangle$ 
and less-iff-word-of [code]:  $\langle p < q \longleftrightarrow \text{word-of } p < \text{word-of } q \rangle$ 
begin

lemma of-class-comm-ring-1:
   $\langle \text{OFCLASS}('a, \text{comm-ring-1-class}) \rangle$ 
by standard (simp-all add: eq-iff-word-of word-of-0 word-of-1
  word-of-add word-of-minus word-of-diff word-of-mult algebra-simps)

lemma of-class-semiring-modulo:
   $\langle \text{OFCLASS}('a, \text{semiring-modulo-class}) \rangle$ 
by standard (simp-all add: eq-iff-word-of word-of-0 word-of-1
  word-of-add word-of-minus word-of-diff word-of-mult word-of-mod word-of-div
algebra-simps
  mod-mult-div-eq)

lemma of-class-equal:
   $\langle \text{OFCLASS}('a, \text{equal-class}) \rangle$ 
by standard (simp add: eq-iff-word-of equal-iff-word-of equal)

lemma of-class-linorder:
   $\langle \text{OFCLASS}('a, \text{linorder-class}) \rangle$ 
by standard (auto simp add: eq-iff-word-of less-eq-iff-word-of less-iff-word-of)

end

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

```

```

n (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⟩
  by (simp add: word-of-0 word-of-1)

```

```

lemma word-of-nat:
  ⟨word-of (of-nat n) = of-nat n⟩
  by (induction n) (simp-all add: word-of-0 word-of-1 word-of-add)

```

```

lemma word-of-numeral [simp]:
  ⟨word-of (numeral n) = numeral n⟩
proof -
  have ⟨word-of (of-nat (numeral n)) = of-nat (numeral n)⟩
    by (simp only: word-of-nat)
  then show ?thesis by simp
qed

```

```

lemma word-of-power:
  ⟨word-of (p ^ n) = word-of p ^ n⟩
  by (induction n) (simp-all add: word-of-1 word-of-mult word-of-numeral)

```

```

lemma even-iff-word-of:
  ⟨2 dvd p ⟷ 2 dvd word-of p⟩ (is ⟨?P ⟷ ?Q⟩)
proof
  assume ?P
  then obtain q where ⟨p = 2 * q⟩ ..
  then show ?Q by (simp add: word-of-mult word-of-numeral)
next
  assume ?Q
  then obtain w where ⟨word-of p = 2 * w⟩ ..
  then have ⟨of-word (word-of p) = of-word (2 * w)⟩
    by simp
  then have ⟨p = 2 * of-word w⟩
    by (simp add: eq-iff-word-of word-of-word word-of-mult word-of-numeral)
  then show ?P
    by simp
qed

```

```

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

```



```

have induct: ⟨ $P\ p$ ⟩ if stable: ⟨ $(\bigwedge p. p\ \text{div}\ 2 = p \implies P\ p)$ ⟩
  and rec: ⟨ $(\bigwedge p\ b. P\ p \implies (\text{of-bool}\ b + 2 * p)\ \text{div}\ 2 = p \implies P\ (\text{of-bool}\ b + 2 * p))$ ⟩
for  $p :: 'a$  and  $P$ 
proof –
  from stable have stable': ⟨ $(\bigwedge p. \text{word-of}\ p\ \text{div}\ 2 = \text{word-of}\ p \implies P\ p)$ ⟩
  by (simp add: eq-iff-word-of word-of-div word-of-numeral)
  from rec have rec': ⟨ $(\bigwedge p\ b. P\ p \implies (\text{of-bool}\ b + 2 * \text{word-of}\ p)\ \text{div}\ 2 = \text{word-of}\ p \implies P\ (\text{of-bool}\ b + 2 * p))$ ⟩
  by (simp add: eq-iff-word-of word-of-add word-of-bool word-of-mult word-of-div word-of-numeral)
  define  $w$  where  $w = \text{word-of}\ p$ 
  then have  $p = \text{of-word}\ w$ 
  by (simp add: of-word-of)
  also have  $P\ (\text{of-word}\ w)$ 
  proof (induction  $w$  rule: bits-induct)
    case (stable  $w$ )
    show  $?case$ 
    by (rule stable') (simp add: word-of-word stable)
  next
    case (rec  $w\ b$ )
    have  $P\ (\text{of-bool}\ b + 2 * \text{of-word}\ w)$ 
    by (rule rec') (simp-all add: word-of-word rec)
    also have  $\text{of-bool}\ b + 2 * \text{of-word}\ w = \text{of-word}\ (\text{of-bool}\ b + 2 * w)$ 
    by (simp add: eq-iff-word-of word-of-word word-of-add word-of-1 word-of-mult word-of-numeral word-of-0)
    finally show  $?case$  .
  qed
  finally show  $P\ p$  .
qed
have ⟨class.semiring-parity-axioms (+) (0:: $'a$ ) (*) 1 (mod)⟩
  by standard (simp-all add: eq-iff-word-of word-of-0 word-of-1 word-of-numeral even-iff-word-of word-of-mod even-iff-mod-2-eq-zero)
  with of-class-semiring-modulo have ⟨OFCLASS(' $a$ , semiring-parity-class)⟩
  by (rule semiring-parity-class.intro)
  moreover have ⟨class.semiring-bits-axioms (+) (–) (0:: $'a$ ) (*) 1 (div) (mod) bit⟩
  apply (standard, fact induct)
  apply (simp-all only: eq-iff-word-of word-of-0 word-of-1 word-of-bool word-of-numeral word-of-add word-of-diff word-of-mult word-of-div word-of-mod word-of-power even-iff-word-of bit-eq-word-of push-bit-take-bit drop-bit-take-bit even-drop-bit-iff-not-bit flip: push-bit-eq-mult drop-bit-eq-div take-bit-eq-mod mask-eq-exp-minus-1)
  apply (auto simp add: ac-simps bit-simps drop-bit-exp-eq push-bit-of-1)
done
ultimately have ⟨OFCLASS(' $a$ , semiring-bits-class)⟩
  by (rule semiring-bits-class.intro)
  moreover have ⟨class.semiring-bit-operations-axioms (+) (–) (*) (1:: $'a$ ) (div)

```

(*mod*) *bit* (*AND*) (*OR*) (*XOR*) *mask* *Bit-Operations.set-bit unset-bit flip-bit push-bit drop-bit take-bit*

by *standard*

(*simp-all add: eq-iff-word-of word-of-push-bit word-of-power*

bit-eq-word-of word-of-and word-of-or word-of-xor word-of-mask word-of-diff

word-of-1 bit-simps

word-of-set-bit set-bit-eq-or word-of-unset-bit word-of-flip-bit flip-bit-eq-xor

word-of-mult

word-of-drop-bit word-of-div word-of-take-bit word-of-mod

flip: mask-eq-exp-minus-1 push-bit-eq-mult drop-bit-eq-div take-bit-eq-mod)

ultimately have $\langle \text{OFCLASS}('a, \text{semiring-bit-operations-class}) \rangle$

by (*rule semiring-bit-operations-class.intro*)

moreover have $\langle \text{OFCLASS}('a, \text{ring-parity-class}) \rangle$

using $\langle \text{OFCLASS}('a, \text{semiring-parity-class}) \rangle$ **by** (*rule ring-parity-class.intro*)

standard

moreover have $\langle \text{class.ring-bit-operations-axioms } (+) (-) (0::'a) (*) 1 \text{ bit uminus } \text{NOT} \rangle$

by *standard* (*simp-all add: eq-iff-word-of word-of-power word-of-numeral*

bit-eq-word-of word-of-diff word-of-1 bit-simps linorder-not-le

word-of-not word-of-0

word-of-minus minus-eq-not-minus-1)

ultimately show $\langle \text{OFCLASS}('a, \text{ring-bit-operations-class}) \rangle$

by (*rule ring-bit-operations-class.intro*)

qed

lemma [*code*]:

$\langle \text{take-bit } n \ a = a \ \text{AND} \ \text{mask } n \rangle$ **for** $a :: 'a$

by (*simp add: eq-iff-word-of word-of-take-bit word-of-and word-of-mask take-bit-eq-mask*)

lemma [*code*]:

$\langle \text{mask } (\text{Suc } n) = \text{push-bit } n \ (1 :: 'a) \ \text{OR} \ \text{mask } n \rangle$

$\langle \text{mask } 0 = (0 :: 'a) \rangle$

by (*simp-all add: eq-iff-word-of word-of-mask word-of-or word-of-push-bit word-of-0 word-of-1 mask-Suc-exp push-bit-of-1*)

lemma [*code*]:

$\langle \text{Bit-Operations.set-bit } n \ w = w \ \text{OR} \ \text{push-bit } n \ 1 \rangle$ **for** $w :: 'a$

by (*simp add: eq-iff-word-of word-of-set-bit word-of-or word-of-push-bit word-of-1 set-bit-eq-or*)

lemma [*code*]:

$\langle \text{unset-bit } n \ w = w \ \text{AND} \ \text{NOT } (\text{push-bit } n \ 1) \rangle$ **for** $w :: 'a$

by (*simp add: eq-iff-word-of word-of-unset-bit word-of-and word-of-not word-of-push-bit word-of-1 unset-bit-eq-and-not*)

lemma [*code*]:

$\langle \text{flip-bit } n \ w = w \ \text{XOR} \ \text{push-bit } n \ 1 \rangle$ **for** $w :: 'a$

by (*simp add: eq-iff-word-of word-of-flip-bit word-of-xor word-of-push-bit word-of-1 flip-bit-eq-xor*)

end

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

lemma *of-word-numeral* [*code-post*]:
 $\langle \text{of-word } (\text{numeral } n) = \text{numeral } n \rangle$
by (*simp add: eq-iff-word-of word-of-word word-of-numeral*)

lemma *of-word-0* [*code-post*]:
 $\langle \text{of-word } 0 = 0 \rangle$
by (*simp add: eq-iff-word-of word-of-0 word-of-word*)

lemma *of-word-1* [*code-post*]:
 $\langle \text{of-word } 1 = 1 \rangle$
by (*simp add: eq-iff-word-of word-of-1 word-of-word*)

Use pretty numerals from integer for pretty printing

lemma *numeral-eq-integer* [*code-unfold*]:
 $\langle \text{numeral } n = \text{of-integer } (\text{numeral } n) \rangle$
by (*simp add: eq-iff-word-of word-of-integer-eq word-of-numeral word-of-integer.rep-eq*)

lemma *neg-numeral-eq-integer* [*code-unfold*]:
 $\langle - \text{numeral } n = \text{of-integer } (- \text{numeral } n) \rangle$
by (*simp add: eq-iff-word-of word-of-integer-eq word-of-minus word-of-numeral word-of-integer.rep-eq*)

end

locale *word-type-copy-misc* = *word-type-copy-more*
opening *constraintless bit-operations-syntax* +
constrains *word-of* :: $\langle 'a::\{\text{ring-bit-operations, equal, linorder}\} \Rightarrow 'b::\text{len word} \rangle$
fixes *size* :: *nat* **and** *set-bits-aux* :: $\langle (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a \rangle$
assumes *size-eq-length*: $\langle \text{size} = \text{LENGTH } ('b::\text{len}) \rangle$
and *msb-iff-word-of* [*code*]: $\langle \text{msb } p \longleftrightarrow \text{msb } (\text{word-of } p) \rangle$
and *lsb-iff-word-of* [*code*]: $\langle \text{lsb } p \longleftrightarrow \text{lsb } (\text{word-of } p) \rangle$
and *size-eq-word-of*: $\langle \text{Nat.size } (p :: 'a) = \text{Nat.size } (\text{word-of } p) \rangle$
and *word-of-generic-set-bit* [*code*]: $\langle \text{word-of } (\text{Generic-set-bit.set-bit } p \ n \ b) =$

Generic-set-bit.set-bit (*word-of p*) *n b*
and *word-of-set-bits*: $\langle \text{word-of } (\text{set-bits } P) = \text{set-bits } P \rangle$
and *word-of-set-bits-aux*: $\langle \text{word-of } (\text{set-bits-aux } P \ n \ p) = \text{Code-Target-Word-Base.set-bits-aux } P \ n \ (\text{word-of } p) \rangle$
begin

lemma *size-eq* [*code*]:
 $\langle \text{Nat.size } p = \text{size} \rangle$ **for** $p :: 'a$
by (*simp add: size-eq-length size-eq-word-of word-size*)

lemma *set-bits-aux-code* [*code*]:
 $\langle \text{set-bits-aux } f \ n \ w =$
(if $n = 0$ *then* w
else $\text{let } n' = n - 1$ *in* $\text{set-bits-aux } f \ n' \ (\text{push-bit } 1 \ w \ \text{OR} \ (\text{if } f \ n' \ \text{then } 1 \ \text{else } 0)) \rangle$
by (*simp add: eq-iff-word-of word-of-set-bits-aux Let-def word-of-mult word-of-or word-of-0 word-of-1 set-bits-aux-rec [of f n]*)

lemma *set-bits-code* [*code*]: $\langle \text{set-bits } P = \text{set-bits-aux } P \ \text{size } 0 \rangle$
by (*simp add: fun-eq-iff eq-iff-word-of word-of-set-bits word-of-set-bits-aux word-of-0 size-eq-length set-bits-conv-set-bits-aux*)

lemma *of-class-lsb*:
 $\langle \text{OFCLASS}('a, \text{lsb-class}) \rangle$
by *standard* (*simp add: fun-eq-iff lsb-iff-word-of even-iff-word-of lsb-odd*)

lemma *of-class-set-bit*:
 $\langle \text{OFCLASS}('a, \text{set-bit-class}) \rangle$
by *standard* (*simp add: eq-iff-word-of word-of-generic-set-bit bit-eq-word-of word-of-power word-of-0 bit-simps linorder-not-le*)

lemma *of-class-bit-comprehension*:
 $\langle \text{OFCLASS}('a, \text{bit-comprehension-class}) \rangle$
by *standard* (*simp add: eq-iff-word-of word-of-set-bits bit-eq-word-of set-bits-bit-eq*)

end

end

Chapter 6

Unsigned words of 64 bits

```
theory Uint64 imports  
  Code-Target-Word-Base Word-Type-Copies  
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.

6.1 Type definition and primitive operations

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

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

global-interpretation *uint64*: *word-type-copy-ring Abs-uint64 Rep-uint64*
by *standard* (*fact zero-uint64.rep-eq one-uint64.rep-eq*
plus-uint64.rep-eq uminus-uint64.rep-eq minus-uint64.rep-eq
times-uint64.rep-eq divide-uint64.rep-eq modulo-uint64.rep-eq
equal-uint64.rep-eq less-eq-uint64.rep-eq less-uint64.rep-eq)⁺

instance proof –

show $\langle \text{OFCLASS}(\text{uint64}, \text{comm-ring-1-class}) \rangle$
by (*rule uint64.of-class-comm-ring-1*)
show $\langle \text{OFCLASS}(\text{uint64}, \text{semiring-modulo-class}) \rangle$
by (*fact uint64.of-class-semiring-modulo*)
show $\langle \text{OFCLASS}(\text{uint64}, \text{equal-class}) \rangle$
by (*fact uint64.of-class-equal*)
show $\langle \text{OFCLASS}(\text{uint64}, \text{linorder-class}) \rangle$
by (*fact uint64.of-class-linorder*)

qed

end

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

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

global-interpretation *uint64*: *word-type-copy-bits Abs-uint64 Rep-uint64 signed-drop-bit-uint64*
by *standard* (*fact bit-uint64.rep-eq not-uint64.rep-eq and-uint64.rep-eq or-uint64.rep-eq*
xor-uint64.rep-eq
mask-uint64.rep-eq push-bit-uint64.rep-eq drop-bit-uint64.rep-eq signed-drop-bit-uint64.rep-eq
take-bit-uint64.rep-eq
set-bit-uint64.rep-eq unset-bit-uint64.rep-eq flip-bit-uint64.rep-eq)+

instance

by (*fact uint64.of-class-ring-bit-operations*)

end

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

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

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

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

context

includes *integer.lifting*

begin

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

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

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
apply *standard*

apply (*simp-all add: uint64-of-nat.rep-eq nat-of-uint64.rep-eq*
uint64-of-int.rep-eq int-of-uint64.rep-eq
Uint64.rep-eq integer-of-uint64.rep-eq integer-eq-iff)

done

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

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


```

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

type uint64 = IntInf.int;

val mask = 0xFFFFFFFFFFFFFFFF : IntInf.int;

val zero = 0 : IntInf.int;

val one = 1 : IntInf.int;

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

fun toInt x = x

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```

end
>
code-reserved SML Uint64

setup <
let
  val polyml64 = LargeWord.wordSize > 63;
  (* PolyML 5.6 has bugs in its Word64 implementation. We test for one such bug
  and refrain
  from using Word64 in that case. Testing is done with dynamic code evaluation
  such that
  the compiler does not choke on the Word64 structure, which need not be present
  in a 32bit
  environment. *)
  val error-msg = Buggy Word64 structure;
  val test-code =
    val - = if Word64.div (0w18446744073709551611 : Word64.word, 0w3) =
0w6148914691236517203 then ()\n ^
    else raise (Fail \ ^ error-msg ^ \);;
    val f = Exn.interruptible-capture (fn () => ML-Compiler.eval ML-Compiler.flags
Position.none (ML-Lex.tokenize test-code))
    val use-Word64 = polyml64 andalso
      (case f () of
        Exn.Res - => true
      | Exn.Exn (e as ERROR m) => if String.isSuffix error-msg m then false else
Exn.reraise e
      | Exn.Exn e => Exn.reraise e)
    ;

  val newline = \n;
  val content =
    structure Uint64 : sig ^ newline ^
      eqtype uint64; ^ newline ^
      val zero : uint64; ^ newline ^
      val one : uint64; ^ newline ^
      val fromInt : IntInf.int -> uint64; ^ newline ^
      val toInt : uint64 -> IntInf.int; ^ newline ^
      val toLarge : uint64 -> LargeWord.word; ^ newline ^
      val fromLarge : LargeWord.word -> uint64 ^ newline ^
      val plus : uint64 -> uint64 -> uint64; ^ newline ^
      val minus : uint64 -> uint64 -> uint64; ^ newline ^
      val times : uint64 -> uint64 -> uint64; ^ newline ^
      val divide : uint64 -> uint64 -> uint64; ^ newline ^
      val modulus : uint64 -> uint64 -> uint64; ^ newline ^
      val negate : uint64 -> uint64; ^ newline ^
      val less-eq : uint64 -> uint64 -> bool; ^ newline ^
      val less : uint64 -> uint64 -> bool; ^ newline ^
      val notb : uint64 -> uint64; ^ newline ^
      val andb : uint64 -> uint64 -> uint64; ^ newline ^

```

```

val orb : uint64 -> uint64 -> uint64; ^ newline ^
val xorb : uint64 -> uint64 -> uint64; ^ newline ^
val shiftl : uint64 -> IntInf.int -> uint64; ^ newline ^
val shiftr : uint64 -> IntInf.int -> uint64; ^ newline ^
val shiftr-signed : uint64 -> IntInf.int -> uint64; ^ newline ^
val set-bit : uint64 -> IntInf.int -> bool -> uint64; ^ newline ^
val test-bit : uint64 -> IntInf.int -> bool; ^ newline ^
end = struct ^ newline ^
^ newline ^
type uint64 = Word64.word; ^ newline ^
^ newline ^
val zero = (0wx0 : uint64); ^ newline ^
^ newline ^
val one = (0wx1 : uint64); ^ newline ^
^ newline ^
fun fromInt x = Word64.fromLargeInt (IntInf.toLarge x); ^ newline ^
^ newline ^
fun toInt x = IntInf.fromLarge (Word64.toLargeInt x); ^ newline ^
^ newline ^
fun fromLarge x = Word64.fromLarge x; ^ newline ^
^ newline ^
fun toLarge x = Word64.toLarge x; ^ newline ^
^ newline ^
fun plus x y = Word64.+(x, y); ^ newline ^
^ newline ^
fun minus x y = Word64.-(x, y); ^ newline ^
^ newline ^
fun negate x = Word64.~(x); ^ newline ^
^ newline ^
fun times x y = Word64.*(x, y); ^ newline ^
^ newline ^
fun divide x y = Word64.div(x, y); ^ newline ^
^ newline ^
fun modulus x y = Word64.mod(x, y); ^ newline ^
^ newline ^
fun less-eq x y = Word64.<=(x, y); ^ newline ^
^ newline ^
fun less x y = Word64.<(x, y); ^ newline ^
^ newline ^
fun set-bit x n b = ^ newline ^
let val mask = Word64.<< (0wx1, Word.fromLargeInt (IntInf.toLarge n)) ^
newline ^
in if b then Word64.orb (x, mask) ^ newline ^
else Word64.andb (x, Word64.notb mask) ^ newline ^
end ^ newline ^
^ newline ^
fun shiftl x n = ^ newline ^
Word64.<< (x, Word.fromLargeInt (IntInf.toLarge n)) ^ newline ^
^ newline ^

```

```

fun shiftr x n = ^ newline ^
  Word64.>> (x, Word.fromLargeInt (IntInf.toLarge n)) ^ newline ^
  ^ newline ^
fun shiftr-signed x n = ^ newline ^
  Word64.~>> (x, Word.fromLargeInt (IntInf.toLarge n)) ^ newline ^
  ^ newline ^
fun test-bit x n = ^ newline ^
  Word64.andb (x, Word64.<<< (0wx1, Word.fromLargeInt (IntInf.toLarge n)))
<> Word64.fromInt 0 ^ newline ^
  ^ newline ^
val notb = Word64.notb ^ newline ^
  ^ newline ^
fun andb x y = Word64.andb(x, y); ^ newline ^
  ^ newline ^
fun orb x y = Word64.orb(x, y); ^ newline ^
  ^ newline ^
fun xorb x y = Word64.xorb(x, y); ^ newline ^
  ^ newline ^
end (*struct Uint64*)
val target-SML64 = SML-word;
in
  (if use-Word64 then Code-Target.set-printings (Code-Symbol.Module (Uint64,
[[Code-Runtime.target, SOME (content, [])]]) else I)
  #> Code-Target.set-printings (Code-Symbol.Module (Uint64, [(target-SML64,
SOME (content, [])])))
end
>

```

```

code-printing code-module Uint64 → (Haskell)
<module Uint64 (Int64, Word64) where

```

```

  import Data.Int(Int64)
  import Data.Word(Word64)
code-reserved Haskell Uint64

```

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

```

code-printing code-module Uint64 → (OCaml)

```

```

<module Uint64 : sig
  val less : int64 -> int64 -> bool
  val less-eq : int64 -> int64 -> bool
  val set-bit : int64 -> Z.t -> bool -> int64
  val shiftl : int64 -> Z.t -> int64
  val shiftr : int64 -> Z.t -> int64
  val shiftr-signed : int64 -> Z.t -> int64
  val test-bit : int64 -> Z.t -> bool
end = struct

```

(* negative numbers have their highest bit set,

```

    so they are greater than positive ones *)
let less x y =
  if Int64.compare x Int64.zero < 0 then
    Int64.compare y Int64.zero < 0 && Int64.compare x y < 0
  else Int64.compare y Int64.zero < 0 || Int64.compare x y < 0;;

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

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

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

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

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

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

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

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

def less(x: Long, y: Long) : Boolean =
  if (x < 0) y < 0 && x < y
  else y < 0 || x < y

def less-eq(x: Long, y: Long) : Boolean =
  if (x < 0) y < 0 && x <= y
  else y < 0 || x <= y

def set-bit(x: Long, n: BigInt, b: Boolean) : Long =
  if (b)
    x | (1L << n.toIntValue)
  else
    x & (1L << n.toIntValue).unary-~

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

```

```

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

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

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

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

```

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

context

includes *bit-operations-syntax*

begin

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

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

lemma *Uint64-code* [code]:

Uint64 $i =$
 (let $i' = i$ AND $0xFFFFFFFFFFFFFFFF$
 in if bit i' 63 then *Uint64-signed* ($i' - 0x10000000000000000$) else *Uint64-signed* i')

including *undefined-transfer integer.lifting unfolding Uint64-signed-def*

apply *transfer*

apply (*subst word-of-int-via-signed*)

apply (*auto simp add: push-bit-of-1 mask-eq-exp-minus-1 word-of-int-via-signed cong del: if-cong*)

done

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

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

unfolding *Uint64-signed-def Uint64-def int-of-integer-symbolic-def word-of-integer-def*
by (*simp add: Abs-uint64-inverse*)

end

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

The new destructor *Rep-uint64'* is executable. As the simplifier is given the [code abstract] equations literally, we cannot implement *Rep-uint64* directly, because that makes `code_simp` loop.

If code generation raises `Match`, some equation probably contains *Rep-uint64*

([code abstract] equations for *uint64* may use *Rep-uint64* because these instances will be folded away.)

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

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

lemma *Rep-uint64'-transfer* [*transfer-rule*]:

rel-fun cr-uint64 (=) (λx. x) Rep-uint64'

unfolding *Rep-uint64'-def* **by**(*rule uint64.rep-transfer*)

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

by *transfer (simp add: set-bits-bit-eq)*

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

lemma *Abs-uint64'-code* [*code*]:

Abs-uint64' x = Uint64 (integer-of-int (uint x))

including *integer.lifting* **by** *transfer simp*

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

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

defines *TR ≡ typerep.TypeRep* **and** *bit0 ≡ STR "Numeral-Type.bit0"*

shows

term-of-class.term-of x =

Code-Evaluation.App (Code-Evaluation.Const (STR "Uint64.uint64.Abs-uint64")
(TR (STR "fun") [TR (STR "Word.word") [TR bit0 [TR bit0 [TR bit0 [TR bit0
[TR bit0 [TR bit0 [TR (STR "Numeral-Type.num1") []]]]]]]], TR (STR "Uint64.uint64")
[]))

(term-of-class.term-of (Rep-uint64' x))

by(*simp add: term-of-anything*)

code-printing

type-constructor *uint64* \rightarrow

(*SML*) *Uint64.uint64* **and**

(*Haskell*) *Uint64.Word64* **and**

(*OCaml*) *int64* **and**

(*Scala*) *Long*

| **constant** *Uint64* \rightarrow

(*SML*) *Uint64.fromInt* **and**

(*Haskell*) (*Prelude.fromInteger - :: Uint64.Word64*) **and**

(*Haskell-Quickcheck*) (*Prelude.fromInteger (Prelude.toInteger -) :: Uint64.Word64*)

and

(*Scala*) *-.longValue*

| **constant** *Uint64-signed* \rightarrow

(*OCaml*) *Z.to'-int64*

| **constant** *0 :: uint64* \rightarrow

(*SML*) *Uint64.zero* **and**

(*Haskell*) (*0 :: Uint64.Word64*) **and**


```

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

```

```

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

```

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

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

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

lemma *div-uint64-code* [*code*]: *x div y* = (if *y* = 0 then 0 else *uint64-div* *x y*)
including *undefined-transfer* **unfolding** *uint64-divmod-def* *uint64-div-def*
by *transfer* (*simp* *add*: *word-div-def*)

lemma *mod-uint64-code* [*code*]: *x mod y* = (if *y* = 0 then *x* else *uint64-mod* *x y*)
including *undefined-transfer* **unfolding** *uint64-mod-def* *uint64-divmod-def*
by *transfer* (*simp* *add*: *word-mod-def*)

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

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

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

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

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

including *undefined-transfer* **unfolding** *uint64-divmod-def* *uint64-sdiv-def* *div0-uint64-def* *mod0-uint64-def*
less-eq-uint64.rep-eq
apply *transfer*
apply (*simp add: divmod-via-sdivmod push-bit-eq-mult*)
done

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

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

unfolding *uint64-sdiv-def* **by**(*simp add: Abs-uint64-inverse*)

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

code-printing

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

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

where [code del]:

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

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

```
bit x n ⟷ n < 64 ∧ uint64-test-bit x (integer-of-nat n)
including undefined-transfer integer.lifting unfolding uint64-test-bit-def
```

by *transfer* (auto dest: bit-imp-le-length)

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

uint64-test-bit $w\ n =$
 (if $n < 0 \vee 63 < n$ then undefined (bit :: $uint64 \Rightarrow -$) $w\ n$ else bit (Rep-uint64
 w) (nat-of-integer n))
unfolding *uint64-test-bit-def* by (simp add: bit-uint64.rep-eq)

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 $x \Rightarrow$ fn $i \Rightarrow$ if $i < 0$ orelse $i \geq 64$ then raise (Fail argument to
uint64'-test'-bit out of bounds) else *Uint64.test'-bit* $x\ i$)

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

where [code del]:

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

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

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

including *undefined-transfer integer.lifting* **unfolding** *uint64-set-bit-def*
by (*transfer*) (auto cong: conj-cong simp add: not-less set-bit-beyond word-size)

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

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

including *undefined-transfer* **unfolding** *uint64-set-bit-def* **by** *transfer simp*

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

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

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

where [code del]:

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

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

including *undefined-transfer integer.lifting* **unfolding** *uint64-shiffl-def*

by *transfer simp*

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

Rep-uint64 (uint64-shiffl w n) =
(if n < 0 ∨ 64 ≤ n then Rep-uint64 (undefined (push-bit :: nat ⇒ uint64 ⇒ -)
w n) else push-bit (nat-of-integer n) (Rep-uint64 w))
including *undefined-transfer unfolding uint64-shiffl-def* **by** *transfer simp*

code-printing constant *uint64-shiffl* \rightarrow

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

definition *uint64-shiftr* :: *uint64 ⇒ integer ⇒ uint64*

where [code del]:

uint64-shiftr x n = (if n < 0 ∨ 64 ≤ n then undefined (drop-bit :: nat ⇒ uint64
⇒ -) x n else drop-bit (nat-of-integer n) x)

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

including *undefined-transfer integer.lifting unfolding uint64-shiftr-def*
by *transfer simp*

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

Rep-uint64 (uint64-shiftr w n) =
(if n < 0 ∨ 64 ≤ n then Rep-uint64 (undefined (drop-bit :: nat ⇒ uint64 ⇒ -)
w n) else drop-bit (nat-of-integer n) (Rep-uint64 w))
including *undefined-transfer unfolding uint64-shiftr-def* **by** *transfer simp*

code-printing constant *uint64-shiftr* \rightarrow

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

definition *uint64-sshiftr* :: *uint64 ⇒ integer ⇒ uint64*

where [code del]:

uint64-sshiftr x n =
(if n < 0 ∨ 64 ≤ n then undefined signed-drop-bit-uint64 n x else signed-drop-bit-uint64
(nat-of-integer n) x)

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

signed-drop-bit-uint64 n x =
(if n < 64 then uint64-sshiftr x (integer-of-nat n) else if bit x 63 then - 1 else 0)

including *undefined-transfer integer.lifting* **unfolding** *uint64-sshiftr-def*
by *transfer (simp add: not-less signed-drop-bit-beyond)*

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

Rep-uint64 (uint64-sshiftr w n) =

(if n < 0 ∨ 64 ≤ n then Rep-uint64 (undefined signed-drop-bit-uint64 n w) else
signed-drop-bit (nat-of-integer n) (Rep-uint64 w))

including *undefined-transfer* **unfolding** *uint64-sshiftr-def* **by** *transfer simp*

code-printing constant *uint64-sshiftr* \rightarrow

(SML) Uint64.shiftr'-signed **and**

(Haskell)

(Prelude.fromInteger (Prelude.toInteger (Data'-Bits.shiftrBounded (Prelude.fromInteger
(Prelude.toInteger -) :: Uint64.Int64 -)) :: Uint64.Word64) and

(OCaml) Uint64.shiftr'-signed **and**

(Scala) Uint64.shiftr'-signed **and**

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

context

includes *bit-operations-syntax*

begin

lemma *uint64-msb-test-bit*: *msb x \longleftrightarrow bit (x :: uint64) 63*

by *transfer (simp add: msb-word-iff-bit)*

lemma *msb-uint64-code* [code]: *msb x \longleftrightarrow uint64-test-bit x 63*

by *(simp add: uint64-test-bit-def uint64-msb-test-bit)*

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

uint64-of-int i = Uint64 (integer-of-int i)

including *integer.lifting* **by** *transfer simp*

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

int-of-uint64 x = int-of-integer (integer-of-uint64 x)

including *integer.lifting* **by** *transfer simp*

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

uint64-of-nat = uint64-of-int ∘ int

by *transfer (simp add: fun-eq-iff)*

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

nat-of-uint64 x = nat-of-integer (integer-of-uint64 x)

unfolding *integer-of-uint64-def* **including** *integer.lifting* **by** *transfer simp*

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)))
  by (simp add: integer-of-uint64-signed-def integer-of-uint64-def)

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 –
  have ⟨integer-of-uint64-signed (n AND 0x7FFFFFFFFFFFFFFF) OR 0x8000000000000000
= Bit-Operations.set-bit 63 (integer-of-uint64-signed (take-bit 63 n))⟩
  by (simp add: take-bit-eq-mask set-bit-eq-or push-bit-eq-mult mask-eq-exp-minus-1)
  moreover have ⟨integer-of-uint64 n = Bit-Operations.set-bit 63 (integer-of-uint64
(take-bit 63 n))⟩ if ⟨bit n 63⟩
  proof (rule bit-eqI)
    fix m
    from that show ⟨bit (integer-of-uint64 n) m = bit (Bit-Operations.set-bit 63
(integer-of-uint64 (take-bit 63 n))) m⟩ for m
    including integer.lifting by transfer (auto simp add: bit-simps dest: bit-imp-le-length)
  qed
  ultimately show ?thesis
  by simp (simp add: integer-of-uint64-signed-def bit-simps)
qed

end

code-printing
  constant integer-of-uint64 ↪
    (SML) Uint64.toInt and
    (Haskell) Prelude.toInteger
| constant integer-of-uint64-signed ↪
    (OCaml) Z.of'-int64 and
    (Scala) BigInt

```

6.3 Quickcheck setup

```

definition uint64-of-natural :: natural ⇒ uint64
where uint64-of-natural x ≡ Uint64 (integer-of-natural x)

```

```

instantiation uint64 :: {random, exhaustive, full-exhaustive} begin
definition random-uint64 ≡ qc-random-cnv uint64-of-natural
definition exhaustive-uint64 ≡ qc-exhaustive-cnv uint64-of-natural
definition full-exhaustive-uint64 ≡ qc-full-exhaustive-cnv uint64-of-natural
instance ..
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") [] .

definition *narrowing-uint64* $d = \text{qc-narrowing-drawn-from } (\text{narrowing-samples } d)$
 d

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

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

instance ..

end

end

Chapter 7

Unsigned words of 32 bits

```
theory Uint32 imports
  Code-Target-Word-Base Word-Type-Copies
begin
```

7.1 Type definition and primitive operations

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

```
global-interpretation uint32: word-type-copy Abs-uint32 Rep-uint32
  using type-definition-uint32 by (rule word-type-copy.intro)
```

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

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

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

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

```
lift-definition zero-uint32 :: uint32 is 0 .
```

```
lift-definition one-uint32 :: uint32 is 1 .
```

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

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

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

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

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

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

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

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

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

```
global-interpretation uint32: word-type-copy-ring Abs-uint32 Rep-uint32
  by standard (fact zero-uint32.rep-eq one-uint32.rep-eq)
```

*plus-uint32.rep-eq uminus-uint32.rep-eq minus-uint32.rep-eq
times-uint32.rep-eq divide-uint32.rep-eq modulo-uint32.rep-eq
equal-uint32.rep-eq less-eq-uint32.rep-eq less-uint32.rep-eq*+

instance proof –

show $\langle OFCLASS(uint32, comm-ring-1-class) \rangle$
by (*rule uint32.of-class-comm-ring-1*)
show $\langle OFCLASS(uint32, semiring-modulo-class) \rangle$
by (*fact uint32.of-class-semiring-modulo*)
show $\langle OFCLASS(uint32, equal-class) \rangle$
by (*fact uint32.of-class-equal*)
show $\langle OFCLASS(uint32, linorder-class) \rangle$
by (*fact uint32.of-class-linorder*)

qed

end

instantiation *uint32* :: *ring-bit-operations*

begin

lift-definition *bit-uint32* :: $\langle uint32 \Rightarrow nat \Rightarrow bool \rangle$ **is** *bit* .

lift-definition *not-uint32* :: $\langle uint32 \Rightarrow uint32 \rangle$ **is** $\langle Bit-Operations.not \rangle$.

lift-definition *and-uint32* :: $\langle uint32 \Rightarrow uint32 \Rightarrow uint32 \rangle$ **is** $\langle Bit-Operations.and \rangle$

.

lift-definition *or-uint32* :: $\langle uint32 \Rightarrow uint32 \Rightarrow uint32 \rangle$ **is** $\langle Bit-Operations.or \rangle$.

lift-definition *xor-uint32* :: $\langle uint32 \Rightarrow uint32 \Rightarrow uint32 \rangle$ **is** $\langle Bit-Operations.xor \rangle$

.

lift-definition *mask-uint32* :: $\langle nat \Rightarrow uint32 \rangle$ **is** *mask* .

lift-definition *push-bit-uint32* :: $\langle nat \Rightarrow uint32 \Rightarrow uint32 \rangle$ **is** *push-bit* .

lift-definition *drop-bit-uint32* :: $\langle nat \Rightarrow uint32 \Rightarrow uint32 \rangle$ **is** *drop-bit* .

lift-definition *signed-drop-bit-uint32* :: $\langle nat \Rightarrow uint32 \Rightarrow uint32 \rangle$ **is** *signed-drop-bit*

.

lift-definition *take-bit-uint32* :: $\langle nat \Rightarrow uint32 \Rightarrow uint32 \rangle$ **is** *take-bit* .

lift-definition *set-bit-uint32* :: $\langle nat \Rightarrow uint32 \Rightarrow uint32 \rangle$ **is** $\langle Bit-Operations.set-bit \rangle$

.

lift-definition *unset-bit-uint32* :: $\langle nat \Rightarrow uint32 \Rightarrow uint32 \rangle$ **is** *unset-bit* .

lift-definition *flip-bit-uint32* :: $\langle nat \Rightarrow uint32 \Rightarrow uint32 \rangle$ **is** *flip-bit* .

global-interpretation *uint32*: *word-type-copy-bits Abs-uint32 Rep-uint32 signed-drop-bit-uint32*

by *standard* (*fact bit-uint32.rep-eq not-uint32.rep-eq and-uint32.rep-eq or-uint32.rep-eq
xor-uint32.rep-eq*

*mask-uint32.rep-eq push-bit-uint32.rep-eq drop-bit-uint32.rep-eq signed-drop-bit-uint32.rep-eq
take-bit-uint32.rep-eq*

set-bit-uint32.rep-eq unset-bit-uint32.rep-eq flip-bit-uint32.rep-eq)+

instance

by (*fact uint32.of-class-ring-bit-operations*)

end

lift-definition *uint32-of-nat* :: $\langle \text{nat} \Rightarrow \text{uint32} \rangle$
is *word-of-nat* .

lift-definition *nat-of-uint32* :: $\langle \text{uint32} \Rightarrow \text{nat} \rangle$
is *unat* .

lift-definition *uint32-of-int* :: $\langle \text{int} \Rightarrow \text{uint32} \rangle$
is *word-of-int* .

lift-definition *int-of-uint32* :: $\langle \text{uint32} \Rightarrow \text{int} \rangle$
is *uint* .

context
includes *integer.lifting*
begin

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

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

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
apply *standard*
apply (*simp-all add: uint32-of-nat.rep-eq nat-of-uint32.rep-eq*
uint32-of-int.rep-eq int-of-uint32.rep-eq
Uint32.rep-eq integer-of-uint32.rep-eq integer-eq-iff)
done

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

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

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

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

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

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

context
includes *lifting-syntax*
begin

```

lemma set-bit-uint32-transfer [transfer-rule]:
  ⟨(cr-uint32 ==> (=) ==> (⟷) ==> cr-uint32) Generic-set-bit.set-bit
  Generic-set-bit.set-bit⟩
  by (simp only: set-bit-eq [abs-def] set-bit-uint32-eq [abs-def]) transfer-prover

```

end

```

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

```

```

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
  by (standard; transfer) simp-all

```

```

instance using uint32.of-class-bit-comprehension
  uint32.of-class-set-bit uint32.of-class-lsb
  by simp-all standard

```

end

7.2 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 set-bit : Word32.word → IntInf.int → bool → Word32.word
  val shiffl : 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 set-bit x n b =
  let val mask = Word32.<< (0wx1, Word.fromLargeInt (IntInf.toLarge n))
  in if b then Word32.orb (x, mask)
  else Word32.andb (x, Word32.notb mask)
  end

```

```

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

```

```

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

```

```

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

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

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

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

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

```

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

```

code-printing code-module Uint32  $\rightarrow$  (OCaml)
<module Uint32 : sig
  val less : int32 -> int32 -> bool
  val less-eq : int32 -> int32 -> bool
  val set-bit : int32 -> Z.t -> bool -> int32
  val shiftl : int32 -> Z.t -> int32
  val shiftr : int32 -> Z.t -> int32
  val shiftr-signed : int32 -> Z.t -> int32
  val test-bit : int32 -> Z.t -> bool
end = struct

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

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

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

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

```

```

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

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

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

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

code-printing code-module Uint32  $\rightarrow$  (Scala)
<object Uint32 {

def less(x: Int, y: Int) : Boolean =
  if (x < 0) y < 0 && x < y
  else y < 0 || x < y

def less-eq(x: Int, y: Int) : Boolean =
  if (x < 0) y < 0 && x <= y
  else y < 0 || x <= y

def set-bit(x: Int, n: BigInt, b: Boolean) : Int =
  if (b)
    x | (1 << n.intValue)
  else
    x & (1 << n.intValue).unary~

def shiffl(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 integer.lifting unfolding Uint32-signed-def
apply transfer
apply (subst word-of-int-via-signed)
apply (auto simp add: push-bit-of-1 mask-eq-exp-minus-1 word-of-int-via-signed)
cong del: if-cong
done

```

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-symbolic i))
unfolding Uint32-signed-def Uint32-def int-of-integer-symbolic-def word-of-integer-def
by(simp add: Abs-uint32-inverse)

```

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'
unfolding Rep-uint32'-def by(rule uint32.rep-transfer)

```

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

by *transfer (simp add: set-bits-bit-eq)*

lift-definition *Abs-uint32' :: 32 word \Rightarrow uint32* **is** $\lambda x :: 32 \text{ word}. x$.

lemma *Abs-uint32'-code* [*code*]:

```

  Abs-uint32' x = Uint32 (integer-of-int (uint x))
including integer.lifting by transfer simp

```

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))
by(simp add: term-of-anything)

```

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

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

lemma *div-uint32-code* [*code*]: *x div y* = (*if y = 0 then 0 else uint32-div x y*)
including *undefined-transfer unfolding uint32-divmod-def uint32-div-def*
by *transfer (simp add: word-div-def)*

lemma *mod-uint32-code* [*code*]: *x mod y* = (*if y = 0 then x else uint32-mod x y*)
including *undefined-transfer unfolding uint32-mod-def uint32-divmod-def*
by *transfer (simp add: word-mod-def)*

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

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

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

lemma *uint32-divmod-code* [*code*]:
uint32-divmod x y =
 (*if 0x80000000 ≤ y then if x < y then (0, x) else (1, x - y)*
 else *if y = 0 then (div0-uint32 x, mod0-uint32 x)*
 else *let q = push-bit 1 (uint32-sdiv (drop-bit 1 x) y);*
 *r = x - q * y*
 in if r ≥ y then (q + 1, r - y) else (q, r))
including *undefined-transfer unfolding uint32-divmod-def uint32-sdiv-def div0-uint32-def*
mod0-uint32-def
less-eq-uint32.rep-eq
apply *transfer*
apply (*simp add: divmod-via-sdivmod push-bit-eq-mult*)
done

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

unfolding *uint32-sdiv-def* **by**(*simp add: Abs-uint32-inverse*)

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

code-printing

constant *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) - '/' -

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

where [code del]:

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

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

bit *x n* $\longleftrightarrow n < 32 \wedge \text{uint32-test-bit } x (\text{integer-of-nat } n)$

including *undefined-transfer integer.lifting* **unfolding** *uint32-test-bit-def*
by (*transfer, simp, transfer, simp*)

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

uint32-test-bit *w n* =
 (if $n < 0 \vee 31 < n$ then *undefined* (*bit* :: *uint32* \Rightarrow -) *w n* else *bit* (*Rep-uint32*
w) (*nat-of-integer* *n*))
unfolding *uint32-test-bit-def* **by**(*simp add: bit-uint32.rep-eq*)

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

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

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

where [code del]:

uint32-set-bit *x n b* =

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

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

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

including undefined-transfer integer.lifting **unfolding** uint32-set-bit-def
by(transfer)(auto cong: conj-cong simp add: not-less set-bit-beyond word-size)

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

Rep-uint32 (uint32-set-bit w n b) =

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

including undefined-transfer **unfolding** uint32-set-bit-def **by** transfer simp

code-printing constant uint32-set-bit \rightarrow

(SML) Uint32.set'-bit **and**

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

(OCaml) Uint32.set'-bit **and**

(Scala) Uint32.set'-bit **and**

(Eval) (fn $w \Rightarrow$ fn $n \Rightarrow$ fn $b \Rightarrow$ if $n < 0$ orelse $32 \leq n$ then raise (Fail
 argument to uint32'-set'-bit out of bounds) else Uint32.set'-bit x n b)

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

where [code del]:

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

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

including undefined-transfer integer.lifting **unfolding** uint32-shiffl-def
by transfer simp

lemma uint32-shiffl-code [code]:

Rep-uint32 (uint32-shiffl w n) =

(if $n < 0 \vee 32 \leq n$ then Rep-uint32 (undefined (push-bit :: nat \Rightarrow uint32 \Rightarrow -)
 w n) else push-bit (nat-of-integer n) (Rep-uint32 w))

including undefined-transfer **unfolding** uint32-shiffl-def
by transfer simp

code-printing constant uint32-shiffl \rightarrow

(SML) Uint32.shiffl **and**

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

(OCaml) Uint32.shiffl **and**

(Scala) Uint32.shiffl **and**

(Eval) (fn $x \Rightarrow$ fn $i \Rightarrow$ if $i < 0$ orelse $i \geq 32$ then raise Fail argument to
 uint32'-shiffl out of bounds else Uint32.shiffl x i)

definition uint32-shiftr :: uint32 \Rightarrow integer \Rightarrow uint32

where [code del]:

$\text{uint32-shiftr } x \ n = (\text{if } n < 0 \vee 32 \leq n \text{ then undefined } (\text{drop-bit} :: \text{nat} \Rightarrow \text{uint32} \Rightarrow -) \ x \ n \text{ else drop-bit } (\text{nat-of-integer } n) \ x)$

lemma *shiftr-uint32-code* [code]: $\text{drop-bit } n \ x = (\text{if } n < 32 \text{ then uint32-shiftr } x \ (\text{integer-of-nat } n) \text{ else } 0)$
including *undefined-transfer integer.lifting unfolding uint32-shiftr-def*
by *transfer simp*

lemma *uint32-shiftr-code* [code]:
 $\text{Rep-uint32 } (\text{uint32-shiftr } w \ n) =$
 $(\text{if } n < 0 \vee 32 \leq n \text{ then Rep-uint32 } (\text{undefined } (\text{drop-bit} :: \text{nat} \Rightarrow \text{uint32} \Rightarrow -) \ w \ n) \text{ else drop-bit } (\text{nat-of-integer } n) \ (\text{Rep-uint32 } w))$
including *undefined-transfer unfolding uint32-shiftr-def by transfer simp*

code-printing constant *uint32-shiftr* \rightarrow
(SML) *Uint32.shiftr* **and**
(Haskell) *Data'-Bits.shiftrBounded* **and**
(OCaml) *Uint32.shiftr* **and**
(Scala) *Uint32.shiftr* **and**
(Eval) $(\text{fn } x \Rightarrow \text{fn } i \Rightarrow \text{if } i < 0 \text{ orelse } i \geq 32 \text{ then raise Fail argument to uint32'-shiftr out of bounds else Uint32.shiftr } x \ i)$

definition *uint32-sshiftr* $:: \text{uint32} \Rightarrow \text{integer} \Rightarrow \text{uint32}$
where [code del]:
 $\text{uint32-sshiftr } x \ n =$
 $(\text{if } n < 0 \vee 32 \leq n \text{ then undefined signed-drop-bit-uint32 } n \ x \text{ else signed-drop-bit-uint32 } (\text{nat-of-integer } n) \ x)$

lemma *sshiftr-uint32-code* [code]:
 $\text{signed-drop-bit-uint32 } n \ x =$
 $(\text{if } n < 32 \text{ then uint32-sshiftr } x \ (\text{integer-of-nat } n) \text{ else if bit } x \ 31 \text{ then } -1 \text{ else } 0)$
including *undefined-transfer integer.lifting unfolding uint32-sshiftr-def*
by *transfer (simp add: not-less signed-drop-bit-beyond)*

lemma *uint32-sshiftr-code* [code]:
 $\text{Rep-uint32 } (\text{uint32-sshiftr } w \ n) =$
 $(\text{if } n < 0 \vee 32 \leq n \text{ then Rep-uint32 } (\text{undefined signed-drop-bit-uint32 } n \ w) \text{ else signed-drop-bit } (\text{nat-of-integer } n) \ (\text{Rep-uint32 } w))$
including *undefined-transfer unfolding uint32-sshiftr-def by transfer simp*

code-printing constant *uint32-sshiftr* \rightarrow
(SML) *Uint32.shiftr'-signed* **and**
(Haskell)
 $(\text{Prelude.fromInteger } (\text{Prelude.toInteger } (\text{Data'-Bits.shiftrBounded } (\text{Prelude.fromInteger } (\text{Prelude.toInteger } -) :: \text{Uint32.Int32}) -)) :: \text{Uint32.Word32})$ **and**
(OCaml) *Uint32.shiftr'-signed* **and**
(Scala) *Uint32.shiftr'-signed* **and**
(Eval) $(\text{fn } x \Rightarrow \text{fn } i \Rightarrow \text{if } i < 0 \text{ orelse } i \geq 32 \text{ then raise Fail argument to uint32'-shiftr'-signed out of bounds else Uint32.shiftr'-signed } x \ i)$

context

includes *bit-operations-syntax*

begin

lemma *uint32-msb-test-bit*: $msb\ x \longleftrightarrow bit\ (x :: uint32)\ 31$
by *transfer (simp add: msb-word-iff-bit)*

lemma *msb-uint32-code* [code]: $msb\ x \longleftrightarrow uint32-test-bit\ x\ 31$
by (*simp add: uint32-test-bit-def uint32-msb-test-bit*)

lemma *uint32-of-int-code* [code]:
 $uint32-of-int\ i = Uint32\ (integer-of-int\ i)$
including *integer.lifting* **by** *transfer simp*

lemma *int-of-uint32-code* [code]:
 $int-of-uint32\ x = int-of-integer\ (integer-of-uint32\ x)$
including *integer.lifting* **by** *transfer simp*

lemma *uint32-of-nat-code* [code]:
 $uint32-of-nat = uint32-of-int \circ int$
by *transfer (simp add: fun-eq-iff)*

lemma *nat-of-uint32-code* [code]:
 $nat-of-uint32\ x = nat-of-integer\ (integer-of-uint32\ x)$
unfolding *integer-of-uint32-def* **including** *integer.lifting* **by** *transfer simp*

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]:
 $integer-of-uint32-signed\ n =$
 $(if\ bit\ n\ 31\ then\ undefined\ integer-of-uint32\ n\ else\ integer-of-int\ (uint\ (Rep-uint32'\ n)))$
by (*simp add: integer-of-uint32-signed-def integer-of-uint32-def*)

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 –

have $\langle integer-of-uint32-signed\ (n\ AND\ 0x7FFFFFFF)\ OR\ 0x80000000 = Bit-Operations.set-bit\ 31\ (integer-of-uint32-signed\ (take-bit\ 31\ n)) \rangle$

by (*simp add: take-bit-eq-mask set-bit-eq-or push-bit-eq-mult mask-eq-exp-minus-1*)

moreover have $\langle integer-of-uint32\ n = Bit-Operations.set-bit\ 31\ (integer-of-uint32\ (take-bit\ 31\ n)) \rangle$ **if** $\langle bit\ n\ 31 \rangle$

proof (*rule bit-eqI*)

```

fix m
  from that show ⟨bit (integer-of-uint32 n) m = bit (Bit-Operations.set-bit 31
(integer-of-uint32 (take-bit 31 n))) m⟩ for m
  including integer.lifting by transfer (auto simp add: bit-simps dest: bit-imp-le-length)
  qed
  ultimately show ?thesis
  by simp (simp add: integer-of-uint32-signed-def bit-simps)
qed

end

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

```

7.3 Quickcheck setup

definition *uint32-of-natural* :: *natural* ⇒ *uint32*

where *uint32-of-natural* *x* ≡ *Uint32* (*integer-of-natural* *x*)

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

definition *random-uint32* ≡ *qc-random-cnv uint32-of-natural*

definition *exhaustive-uint32* ≡ *qc-exhaustive-cnv uint32-of-natural*

definition *full-exhaustive-uint32* ≡ *qc-full-exhaustive-cnv uint32-of-natural*

instance ..

end

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

interpretation *quickcheck-narrowing-samples*

λ*i*. let *x* = *Uint32* *i* in (*x*, *0xFFFFFFFF* - *x*) 0

Typerep.*Typerep* (*STR* "*Uint32.uint32*") [] .

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

declare [[*code drop: partial-term-of* :: *uint32* *itself* ⇒ -]]

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

instance ..

end

end

Chapter 8

Unsigned words of 16 bits

```
theory Uint16 imports  
  Code-Target-Word-Base Word-Type-Copies  
begin
```

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

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

8.1 Type definition and primitive operations

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

lift-definition *uminus-uint16* :: $\langle \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** *uminus* .
lift-definition *minus-uint16* :: $\langle \text{uint16} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** $\langle (-) \rangle$.
lift-definition *times-uint16* :: $\langle \text{uint16} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** $\langle (*) \rangle$.
lift-definition *divide-uint16* :: $\langle \text{uint16} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** $\langle (div) \rangle$.
lift-definition *modulo-uint16* :: $\langle \text{uint16} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** $\langle (mod) \rangle$.
lift-definition *equal-uint16* :: $\langle \text{uint16} \Rightarrow \text{uint16} \Rightarrow \text{bool} \rangle$ **is** $\langle \text{HOL.equal} \rangle$.
lift-definition *less-eq-uint16* :: $\langle \text{uint16} \Rightarrow \text{uint16} \Rightarrow \text{bool} \rangle$ **is** $\langle (\leq) \rangle$.
lift-definition *less-uint16* :: $\langle \text{uint16} \Rightarrow \text{uint16} \Rightarrow \text{bool} \rangle$ **is** $\langle (<) \rangle$.

global-interpretation *uint16*: *word-type-copy-ring Abs-uint16 Rep-uint16*

by *standard* (*fact* *zero-uint16.rep-eq one-uint16.rep-eq*
plus-uint16.rep-eq uminus-uint16.rep-eq minus-uint16.rep-eq
times-uint16.rep-eq divide-uint16.rep-eq modulo-uint16.rep-eq
equal-uint16.rep-eq less-eq-uint16.rep-eq less-uint16.rep-eq)⁺

instance proof –

show $\langle \text{OFCLASS}(\text{uint16}, \text{comm-ring-1-class}) \rangle$
by (*rule* *uint16.of-class-comm-ring-1*)
show $\langle \text{OFCLASS}(\text{uint16}, \text{semiring-modulo-class}) \rangle$
by (*fact* *uint16.of-class-semiring-modulo*)
show $\langle \text{OFCLASS}(\text{uint16}, \text{equal-class}) \rangle$
by (*fact* *uint16.of-class-equal*)
show $\langle \text{OFCLASS}(\text{uint16}, \text{linorder-class}) \rangle$
by (*fact* *uint16.of-class-linorder*)

qed

end

instantiation *uint16* :: *ring-bit-operations*

begin

lift-definition *bit-uint16* :: $\langle \text{uint16} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$ **is** *bit* .
lift-definition *not-uint16* :: $\langle \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** $\langle \text{Bit-Operations.not} \rangle$.
lift-definition *and-uint16* :: $\langle \text{uint16} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** $\langle \text{Bit-Operations.and} \rangle$.
lift-definition *or-uint16* :: $\langle \text{uint16} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** $\langle \text{Bit-Operations.or} \rangle$.
lift-definition *xor-uint16* :: $\langle \text{uint16} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** $\langle \text{Bit-Operations.xor} \rangle$.
lift-definition *mask-uint16* :: $\langle \text{nat} \Rightarrow \text{uint16} \rangle$ **is** *mask* .
lift-definition *push-bit-uint16* :: $\langle \text{nat} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** *push-bit* .
lift-definition *drop-bit-uint16* :: $\langle \text{nat} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** *drop-bit* .
lift-definition *signed-drop-bit-uint16* :: $\langle \text{nat} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** *signed-drop-bit* .
lift-definition *take-bit-uint16* :: $\langle \text{nat} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** *take-bit* .
lift-definition *set-bit-uint16* :: $\langle \text{nat} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** $\langle \text{Bit-Operations.set-bit} \rangle$.
lift-definition *unset-bit-uint16* :: $\langle \text{nat} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** *unset-bit* .
lift-definition *flip-bit-uint16* :: $\langle \text{nat} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** *flip-bit* .

global-interpretation *uint16*: *word-type-copy-bits Abs-uint16 Rep-uint16 signed-drop-bit-uint16*
by *standard* (*fact bit-uint16.rep-eq not-uint16.rep-eq and-uint16.rep-eq or-uint16.rep-eq*
xor-uint16.rep-eq
mask-uint16.rep-eq push-bit-uint16.rep-eq drop-bit-uint16.rep-eq signed-drop-bit-uint16.rep-eq
take-bit-uint16.rep-eq
set-bit-uint16.rep-eq unset-bit-uint16.rep-eq flip-bit-uint16.rep-eq)+

instance

by (*fact uint16.of-class-ring-bit-operations*)

end

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

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

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

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

context

includes *integer.lifting*

begin

lift-definition *Uuint16* :: $\langle \text{integer} \Rightarrow \text{uint16} \rangle$
is *word-of-int* .

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

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 Uuint16 integer-of-uint16
apply *standard*
apply (*simp-all add: uint16-of-nat.rep-eq nat-of-uint16.rep-eq*
uint16-of-int.rep-eq int-of-uint16.rep-eq
Uuint16.rep-eq integer-of-uint16.rep-eq integer-eq-iff)
done

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

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

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

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

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

definition *set-bit-uint16* :: $\langle \text{uint16} \Rightarrow \text{nat} \Rightarrow \text{bool} \Rightarrow \text{uint16} \rangle$

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

context

includes *lifting-syntax*

begin

lemma *set-bit-uint16-transfer* [*transfer-rule*]:

$\langle (\text{cr-uint16} \ \text{====} \ (=) \ \text{====} \ (\longleftrightarrow) \ \text{====} \ \text{cr-uint16}) \ \text{Generic-set-bit.set-bit Generic-set-bit.set-bit} \rangle$

by (*simp only: set-bit-eq [abs-def] set-bit-uint16-eq [abs-def]*) *transfer-prover*

end

lift-definition *set-bits-uint16* :: $\langle (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{uint16} \rangle$ **is** *set-bits* .

lift-definition *set-bits-aux-uint16* :: $\langle (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow \text{uint16} \Rightarrow \text{uint16} \rangle$ **is** *set-bits-aux* .

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*

by (*standard; transfer*) *simp-all*

instance using *uint16.of-class-bit-comprehension*

uint16.of-class-set-bit uint16.of-class-lsb

by *simp-all standard*

end

8.2 Code setup

code-printing code-module *Uint16* \rightarrow (*SML-word*)

$\langle (* \ \text{Test that words can handle numbers between 0 and 15} \ *) \rangle$

val - = *if* 4 <= *Word.wordSize* *then* () *else raise* (*Fail (wordSize less than 4)*);

structure *Uint16* : *sig*

val *set-bit* : *Word16.word* \rightarrow *IntInf.int* \rightarrow *bool* \rightarrow *Word16.word*

val *shiffl* : *Word16.word* \rightarrow *IntInf.int* \rightarrow *Word16.word*

val *shiftr* : *Word16.word* \rightarrow *IntInf.int* \rightarrow *Word16.word*

val *shiftr-signed* : *Word16.word* \rightarrow *IntInf.int* \rightarrow *Word16.word*

val *test-bit* : *Word16.word* \rightarrow *IntInf.int* \rightarrow *bool*

end = *struct*

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

```

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

fun 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 → (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 → (Scala)
⟨object Uint16 {

def set-bit(x: scala.Char, n: BigInt, b: Boolean) : scala.Char =
  if (b)
    (x | (1.toChar <<< n.intValue)).toChar
  else
    (x & (1.toChar <<< n.intValue).unary~).toChar

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

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

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

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

} /* object Uint16 */
code-reserved Scala Uint16

```

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

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

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

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

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

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

rel-fun cr-uint16 (=) (λx. x) Rep-uint16'

unfolding *Rep-uint16'-def* **by**(`rule uint16.rep-transfer`)

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

by *transfer (simp add: set-bits-bit-eq)*

lift-definition *Abs-uint16'* `:: 16 word ⇒ uint16` **is** `λx :: 16 word. x .`

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

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

including *integer.lifting* **by** *transfer simp*

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)*)

by(*simp add: term-of-anything*)

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

unfolding *Uint16-def int-of-integer-symbolic-def* **by**(*simp add: Abs-uint16-inverse*)

code-printing

type-constructor *uint16* `→`

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

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

(*Scala*) *scala.Char*

| **constant** *Uint16* `→`

(*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 →
  (SML-word) (Word16.fromInt 0) and
  (Haskell) (0 :: Uint16.Word16) and
  (Scala) 0
| constant 1 :: uint16 →
  (SML-word) (Word16.fromInt 1) and
  (Haskell) (1 :: Uint16.Word16) and
  (Scala) 1
| constant plus :: uint16 ⇒ - ⇒ - →
  (SML-word) Word16.+ ((-), (-)) and
  (Haskell) infixl 6 + and
  (Scala) (- +/ -).toChar
| constant uminus :: uint16 ⇒ - →
  (SML-word) Word16.~ and
  (Haskell) negate and
  (Scala) (- -).toChar
| constant minus :: uint16 ⇒ - →
  (SML-word) Word16.- ((-), (-)) and
  (Haskell) infixl 6 - and
  (Scala) (- -/ -).toChar
| constant times :: uint16 ⇒ - ⇒ - →
  (SML-word) Word16.* ((-), (-)) and
  (Haskell) infixl 7 * and
  (Scala) (- */ -).toChar
| constant HOL.equal :: uint16 ⇒ - ⇒ bool →
  (SML-word) !((- : Word16.word) = -) and
  (Haskell) infix 4 == and
  (Scala) infixl 5 ==
| class-instance uint16 :: equal → (Haskell) -
| constant less-eq :: uint16 ⇒ - ⇒ bool →
  (SML-word) Word16.<= ((-), (-)) and
  (Haskell) infix 4 <= and
  (Scala) infixl 4 <=
| constant less :: uint16 ⇒ - ⇒ bool →
  (SML-word) Word16.< ((-), (-)) and
  (Haskell) infix 4 < and
  (Scala) infixl 4 <
| constant Bit-Operations.not :: uint16 ⇒ - →
  (SML-word) Word16.notb and
  (Haskell) Data'-Bits.complement and
  (Scala) -.unary'~.toChar
| constant Bit-Operations.and :: uint16 ⇒ - →
  (SML-word) Word16.andb ((-), (-)) and
  (Haskell) infixl 7 Data-Bits.& and
  (Scala) (- & -).toChar
| constant Bit-Operations.or :: uint16 ⇒ - →

```

```

(SML-word) Word16.orb ((-),/ (-)) and
(Haskell) infixl 5 Data-Bits..|. and
(Scala) (- | -).toChar
| constant Bit-Operations.xor :: uint16 ⇒ - →
(SML-word) Word16.xorb ((-),/ (-)) and
(Haskell) Data'-Bits.xor and
(Scala) (- ^ -).toChar

```

```

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

```

```

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

```

context includes *undefined-transfer* **begin**

```

lemma div-uint16-code [code]: x div y = (if y = 0 then 0 else uint16-div x y)
unfolding uint16-div-def by transfer (simp add: word-div-def)

```

```

lemma mod-uint16-code [code]: x mod y = (if y = 0 then x else uint16-mod x y)
unfolding uint16-mod-def by transfer (simp add: word-mod-def)

```

```

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)
unfolding uint16-div-def by transfer simp

```

```

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)
unfolding uint16-mod-def by transfer simp

```

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

```

```

definition uint16-test-bit :: uint16 ⇒ integer ⇒ bool
where [code del]:

```


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

lemma *test-bit-uint16-code* [code]:
 bit $x\ n \iff n < 16 \wedge$ *uint16-test-bit* x (integer-of-nat n)
including *undefined-transfer integer.lifting unfolding uint16-test-bit-def*
by (*transfer, simp, transfer, simp*)

lemma *uint16-test-bit-code* [code]:
uint16-test-bit $w\ n =$
 (if $n < 0 \vee 15 < n$ then undefined (bit :: *uint16* \Rightarrow -) $w\ n$ else bit (*Rep-uint16*
 w) (nat-of-integer n))
unfolding *uint16-test-bit-def* **by** (*simp add: bit-uint16.rep-eq*)

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

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

lemma *set-bit-uint16-code* [code]:
set-bit $x\ n\ b =$ (if $n < 16$ then *uint16-set-bit* x (integer-of-nat n) b else x)
including *undefined-transfer integer.lifting unfolding uint16-set-bit-def*
by(*transfer*)(*auto cong: conj-cong simp add: not-less set-bit-beyond word-size*)

lemma *uint16-set-bit-code* [code]:
Rep-uint16 (*uint16-set-bit* $w\ n\ b$) =
 (if $n < 0 \vee 15 < n$ then *Rep-uint16* (undefined (*set-bit* :: *uint16* \Rightarrow -) $w\ n\ b$)
 else *set-bit* (*Rep-uint16* w) (nat-of-integer n) b)
including *undefined-transfer unfolding uint16-set-bit-def* **by** *transfer simp*

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

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

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

including *undefined-transfer integer.lifting unfolding uint16-shiffl-def*
by *transfer simp*

lemma *uint16-shiffl-code* [code]:
 $Rep-uint16 (uint16-shiffl w n) =$
(if $n < 0 \vee 16 \leq n$ then $Rep-uint16 (undefined (push-bit :: nat \Rightarrow uint16 \Rightarrow -)$
 $w n)$
else $push-bit (nat-of-integer n) (Rep-uint16 w)$
including *undefined-transfer unfolding uint16-shiffl-def*
by *transfer simp*

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

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

lemma *shiftr-uint16-code* [code]: $drop-bit n x = (if n < 16$ then *uint16-shiftr x*
(integer-of-nat n) else 0)
including *undefined-transfer integer.lifting unfolding uint16-shiftr-def*
by *transfer simp*

lemma *uint16-shiftr-code* [code]:
 $Rep-uint16 (uint16-shiftr w n) =$
(if $n < 0 \vee 16 \leq n$ then $Rep-uint16 (undefined (drop-bit :: nat \Rightarrow uint16 \Rightarrow -)$
 $w n)$
else $drop-bit (nat-of-integer n) (Rep-uint16 w)$
including *undefined-transfer unfolding uint16-shiftr-def* **by** *transfer simp*

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

definition *uint16-sshiftr* :: *uint16* \Rightarrow *integer* \Rightarrow *uint16*
where [code del]:
 $uint16-sshiftr x n =$
(if $n < 0 \vee 16 \leq n$ then $undefined signed-drop-bit-uint16 n x$ else $signed-drop-bit-uint16$
 $(nat-of-integer n) x$)

lemma *sshiftr-uint16-code* [code]:
 $signed-drop-bit-uint16 n x =$
(if $n < 16$ then $uint16-sshiftr x (integer-of-nat n)$ else if bit x 15 then -1 else 0)
including *undefined-transfer integer.lifting unfolding uint16-sshiftr-def*
by *transfer (simp add: not-less signed-drop-bit-beyond word-size)*

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

Rep-uint16 (uint16-sshiftr w n) =
(if n < 0 ∨ 16 ≤ n then Rep-uint16 (undefined signed-drop-bit-uint16 n w)
else signed-drop-bit (nat-of-integer n) (Rep-uint16 w))
including *undefined-transfer* **unfolding** *uint16-sshiftr-def*
by *transfer simp*

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

by *transfer (simp add: msb-word-iff-bit)*

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

by *(simp add: uint16-test-bit-def uint16-msb-test-bit)*

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

including *integer.lifting* **by** *transfer simp*

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

int-of-uint16 x = int-of-integer (integer-of-uint16 x)
by *(simp add: int-of-uint16.rep-eq integer-of-uint16-def)*

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

uint16-of-nat = uint16-of-int ∘ int
by *transfer (simp add: fun-eq-iff)*

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

nat-of-uint16 x = nat-of-integer (integer-of-uint16 x)
unfolding *integer-of-uint16-def* **including** *integer.lifting* **by** *transfer simp*

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

integer-of-uint16 n = integer-of-int (uint (Rep-uint16' n))
unfolding *integer-of-uint16-def* **by** *transfer auto*

code-printing

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

8.3 Quickcheck setup

definition *uint16-of-natural* :: $natural \Rightarrow uint16$

where *wint16-of-natural* $x \equiv \text{Uint16 } (\text{integer-of-natural } x)$

instantiation *wint16* :: {*random, exhaustive, full-exhaustive*} **begin**

definition *random-wint16* $\equiv \text{qc-random-cnv } \text{wint16-of-natural}$

definition *exhaustive-wint16* $\equiv \text{qc-exhaustive-cnv } \text{wint16-of-natural}$

definition *full-exhaustive-wint16* $\equiv \text{qc-full-exhaustive-cnv } \text{wint16-of-natural}$

instance ..

end

instantiation *wint16* :: *narrowing* **begin**

interpretation *quickcheck-narrowing-samples*

$\lambda i. \text{let } x = \text{Uint16 } i \text{ in } (x, 0xFFFF - x) \quad 0$

$\text{Typerep. Typerep } (\text{STR } \text{"Uint16.wint16"}) \quad []$.

definition *narrowing-wint16* $d = \text{qc-narrowing-drawn-from } (\text{narrowing-samples } d)$

declare [[*code drop: partial-term-of* :: *wint16* *itself* \Rightarrow -]]

lemmas *partial-term-of-wint16* [*code*] = *partial-term-of-code*

instance ..

end

end

Chapter 9

Unsigned words of 8 bits

```
theory Uint8 imports  
  Code-Target-Word-Base Word-Type-Copies  
begin
```

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

9.1 Type definition and primitive operations

```
typedef uint8 =  $\langle UNIV :: 8 \text{ word set} \rangle ..$ 
```

```
global-interpretation uint8: word-type-copy Abs-uint8 Rep-uint8  
  using type-definition-uint8 by (rule word-type-copy.intro)
```

```
setup-lifting type-definition-uint8
```

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

```
declare Quotient-uint8 [transfer-rule]
```

```
instantiation uint8 ::  $\langle \{ \text{comm-ring-1, semiring-modulo, equal, linorder} \} \rangle$   
begin
```

```
lift-definition zero-uint8 :: uint8 is  $0$  .
```

```
lift-definition one-uint8 :: uint8 is  $1$  .
```

```
lift-definition plus-uint8 ::  $\langle uint8 \Rightarrow uint8 \Rightarrow uint8 \rangle$  is  $\langle (+) \rangle$  .
```

```
lift-definition uminus-uint8 ::  $\langle uint8 \Rightarrow uint8 \rangle$  is uminus .
```

```
lift-definition minus-uint8 ::  $\langle uint8 \Rightarrow uint8 \Rightarrow uint8 \rangle$  is  $\langle (-) \rangle$  .
```

```
lift-definition times-uint8 ::  $\langle uint8 \Rightarrow uint8 \Rightarrow uint8 \rangle$  is  $\langle (*) \rangle$  .
```

```
lift-definition divide-uint8 ::  $\langle uint8 \Rightarrow uint8 \Rightarrow uint8 \rangle$  is  $\langle (div) \rangle$  .
```

```
lift-definition modulo-uint8 ::  $\langle uint8 \Rightarrow uint8 \Rightarrow uint8 \rangle$  is  $\langle (mod) \rangle$  .
```

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

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

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

global-interpretation *uint8*: *word-type-copy-ring Abs-uint8 Rep-uint8*

by standard (*fact zero-uint8.rep-eq one-uint8.rep-eq plus-uint8.rep-eq uminus-uint8.rep-eq minus-uint8.rep-eq times-uint8.rep-eq divide-uint8.rep-eq modulo-uint8.rep-eq equal-uint8.rep-eq less-eq-uint8.rep-eq less-uint8.rep-eq*)+

instance proof –

show $\langle \text{OFCLASS}(\text{uint8}, \text{comm-ring-1-class}) \rangle$
by (*rule uint8.of-class-comm-ring-1*)
show $\langle \text{OFCLASS}(\text{uint8}, \text{semiring-modulo-class}) \rangle$
by (*fact uint8.of-class-semiring-modulo*)
show $\langle \text{OFCLASS}(\text{uint8}, \text{equal-class}) \rangle$
by (*fact uint8.of-class-equal*)
show $\langle \text{OFCLASS}(\text{uint8}, \text{linorder-class}) \rangle$
by (*fact uint8.of-class-linorder*)

qed

end

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

begin

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

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

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

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

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

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

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

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

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

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

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

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

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

global-interpretation *uint8*: *word-type-copy-bits Abs-uint8 Rep-uint8 signed-drop-bit-uint8*

by standard (*fact bit-uint8.rep-eq not-uint8.rep-eq and-uint8.rep-eq or-uint8.rep-eq xor-uint8.rep-eq*

mask-uint8.rep-eq push-bit-uint8.rep-eq drop-bit-uint8.rep-eq signed-drop-bit-uint8.rep-eq take-bit-uint8.rep-eq

set-bit-uint8.rep-eq unset-bit-uint8.rep-eq flip-bit-uint8.rep-eq)+

instance

by (*fact uint8.of-class-ring-bit-operations*)

end

lift-definition *wint8-of-nat* :: $\langle \text{nat} \Rightarrow \text{wint8} \rangle$
is *word-of-nat* .

lift-definition *nat-of-wint8* :: $\langle \text{wint8} \Rightarrow \text{nat} \rangle$
is *unat* .

lift-definition *wint8-of-int* :: $\langle \text{int} \Rightarrow \text{wint8} \rangle$
is *word-of-int* .

lift-definition *int-of-wint8* :: $\langle \text{wint8} \Rightarrow \text{int} \rangle$
is *wint* .

context
includes *integer.lifting*
begin

lift-definition *Uint8* :: $\langle \text{integer} \Rightarrow \text{wint8} \rangle$
is *word-of-int* .

lift-definition *integer-of-wint8* :: $\langle \text{wint8} \Rightarrow \text{integer} \rangle$
is *wint* .

end

global-interpretation *wint8*: *word-type-copy-more Abs-wint8 Rep-wint8 signed-drop-bit-wint8*
wint8-of-nat nat-of-wint8 wint8-of-int int-of-wint8 Uint8 integer-of-wint8
apply *standard*
apply (*simp-all add: wint8-of-nat.rep-eq nat-of-wint8.rep-eq*
wint8-of-int.rep-eq int-of-wint8.rep-eq
Uint8.rep-eq integer-of-wint8.rep-eq integer-eq-iff)
done

instantiation *wint8* :: $\{ \text{size}, \text{msb}, \text{lsb}, \text{set-bit}, \text{bit-comprehension} \}$
begin

lift-definition *size-wint8* :: $\langle \text{wint8} \Rightarrow \text{nat} \rangle$ **is** *size* .

lift-definition *msb-wint8* :: $\langle \text{wint8} \Rightarrow \text{bool} \rangle$ **is** *msb* .

lift-definition *lsb-wint8* :: $\langle \text{wint8} \Rightarrow \text{bool} \rangle$ **is** *lsb* .

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

definition *set-bit-wint8* :: $\langle \text{wint8} \Rightarrow \text{nat} \Rightarrow \text{bool} \Rightarrow \text{wint8} \rangle$
where *set-bit-wint8-eq*: $\langle \text{set-bit-wint8 } a \ n \ b = (\text{if } b \text{ then } \text{Bit-Operations.set-bit else } \text{unset-bit}) \ n \ a \rangle$

context
includes *lifting-syntax*
begin

```

lemma set-bit-uint8-transfer [transfer-rule]:
  ⟨(cr-uint8 ==> (=) ==> (←→) ==> cr-uint8) Generic-set-bit.set-bit
  Generic-set-bit.set-bit⟩
  by (simp only: set-bit-eq [abs-def] set-bit-uint8-eq [abs-def]) transfer-prover

```

end

```

lift-definition set-bits-uint8 :: ⟨(nat ⇒ bool) ⇒ uint8⟩ is set-bits .

```

```

lift-definition set-bits-aux-uint8 :: ⟨(nat ⇒ bool) ⇒ nat ⇒ uint8 ⇒ uint8⟩ is
set-bits-aux .

```

```

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
by (standard; transfer) simp-all

```

```

instance using uint8.of-class-bit-comprehension
uint8.of-class-set-bit uint8.of-class-lsb
by simp-all standard

```

end

9.2 Code setup

```

code-printing code-module Uint8 ↪ (SML)
⟨(* Test that words can handle numbers between 0 and 3 *)
val - = if 3 <= Word.wordSize then () else raise (Fail (wordSize less than 3));

```

```

structure Uint8 : sig
  val set-bit : Word8.word -> IntInf.int -> bool -> Word8.word
  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 set-bit x n b =
  let val mask = Word8.<< (0wx1, Word.fromLargeInt (IntInf.toLarge n))
  in if b then Word8.orb (x, mask)
  else Word8.andb (x, Word8.notb mask)
end

```

```

fun shiftl x n =
  Word8.<< (x, Word.fromLargeInt (IntInf.toLarge n))

```

```

fun shiftr x n =
  Word8.>> (x, Word.fromLargeInt (IntInf.toLarge n))

```

```

fun shiftr-signed x n =

```



```

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

fun test-bit x n =
  Word8.andb (x, Word8.<<< (0wx1, Word.fromLargeInt (IntInf.toLarge n))) <>
  Word8.fromInt 0

end; (* struct Uint8 *)
code-reserved SML Uint8

code-printing code-module Uint8  $\rightarrow$  (Haskell)
<module Uint8(Int8, Word8) where

  import Data.Int(Int8)
  import Data.Word(Word8)>
code-reserved Haskell Uint8

```

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

```

code-printing code-module Uint8  $\rightarrow$  (Scala)
<object Uint8 {

  def less(x: Byte, y: Byte) : Boolean =
    if (x < 0) y < 0 && x < y
    else y < 0 || x < y

  def less-eq(x: Byte, y: Byte) : Boolean =
    if (x < 0) y < 0 && x <= y
    else y < 0 || x <= y

  def set-bit(x: Byte, n: BigInt, b: Boolean) : Byte =
    if (b)
      (x | (1 << n.intValue)).toByte
    else
      (x & (1 << n.intValue).unary~).toByte

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

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

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

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

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

```

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

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

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

To convert 8 word values into $uint8$, use $Abs-uint8'$.

definition $Rep-uint8'$ **where** [simp]: $Rep-uint8' = Rep-uint8$

lemma $Rep-uint8'$ -transfer [transfer-rule]:

rel-fun cr-uint8 (=) ($\lambda x. x$) $Rep-uint8'$

unfolding $Rep-uint8'$ -def **by**(rule $uint8.rep-transfer$)

lemma $Rep-uint8'$ -code [code]: $Rep-uint8' x = (BITS\ n.\ bit\ x\ n)$

by transfer (simp add: set-bits-bit-eq)

lift-definition $Abs-uint8' :: 8\ word \Rightarrow uint8$ **is** $\lambda x :: 8\ word.\ x$.

lemma $Abs-uint8'$ -code [code]: $Abs-uint8' x = Uint8\ (integer-of-int\ (uint\ x))$

including *integer.lifting* **by** transfer simp

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 (STR
"Numeral-Type.num1") []]]], TR (STR "Uint8.uint8") []))
(term-of-class.term-of (Rep-uint8' x))

by(simp add: term-of-anything)

lemma $Uin8-code$ [code]: $Rep-uint8\ (Uint8\ i) = word-of-int\ (int-of-integer-symbolic\ i)$

unfolding $Uin8-def\ int-of-integer-symbolic-def$ **by**(simp add: $Abs-uint8-inverse$)

code-printing type-constructor $uint8 \rightarrow$

(SML) $Word8.word$ **and**

(Haskell) $Uin8.Word8$ **and**

(Scala) $Byte$

| **constant** $Uin8 \rightarrow$

(SML) $Word8.fromLargeInt\ (IntInf.toLarge\ -)$ **and**

(Haskell) $(Prelude.fromInteger\ - :: Uin8.Word8)$ **and**

(Haskell-Quickcheck) $(Prelude.fromInteger\ (Prelude.toInteger\ -) :: Uin8.Word8)$

and

(Scala) $-.byteValue$

| **constant** $0 :: uint8 \rightarrow$

(SML) $(Word8.fromInt\ 0)$ **and**

```

(Haskell) (0 :: Uint8.Word8) and
(Scala) 0.toByte
| constant 1 :: uint8 →
(SML) (Word8.fromInt 1) and
(Haskell) (1 :: Uint8.Word8) and
(Scala) 1.toByte
| constant plus :: uint8 ⇒ - ⇒ - →
(SML) Word8.+ ((-), (-)) and
(Haskell) infixl 6 + and
(Scala) (- +/ -).toByte
| constant uminus :: uint8 ⇒ - →
(SML) Word8.~ and
(Haskell) negate and
(Scala) (- -).toByte
| constant minus :: uint8 ⇒ - →
(SML) Word8.- ((-), (-)) and
(Haskell) infixl 6 - and
(Scala) (- -/ -).toByte
| constant times :: uint8 ⇒ - ⇒ - →
(SML) Word8.* ((-), (-)) and
(Haskell) infixl 7 * and
(Scala) (- */ -).toByte
| constant HOL.equal :: uint8 ⇒ - ⇒ bool →
(SML) !((- : Word8.word) = -) and
(Haskell) infix 4 == and
(Scala) infixl 5 ==
| class-instance uint8 :: equal → (Haskell) -
| constant less-eq :: uint8 ⇒ - ⇒ bool →
(SML) Word8.<= ((-), (-)) and
(Haskell) infix 4 <= and
(Scala) Uint8.less'-eq
| constant less :: uint8 ⇒ - ⇒ bool →
(SML) Word8.< ((-), (-)) and
(Haskell) infix 4 < and
(Scala) Uint8.less
| constant Bit-Operations.not :: uint8 ⇒ - →
(SML) Word8.notb and
(Haskell) Data'-Bits.complement and
(Scala) -.unary'~.toByte
| constant Bit-Operations.and :: uint8 ⇒ - →
(SML) Word8.andb ((-),/ (-)) and
(Haskell) infixl 7 Data-Bits.&. and
(Scala) (- & -).toByte
| constant Bit-Operations.or :: uint8 ⇒ - →
(SML) Word8.orb ((-),/ (-)) and
(Haskell) infixl 5 Data-Bits.|. and
(Scala) (- | -).toByte
| constant Bit-Operations.xor :: uint8 ⇒ - →
(SML) Word8.xorb ((-),/ (-)) and

```

(Haskell) *Data'-Bits.xor* and
 (Scala) *(- ^ -).toByte*

definition *uint8-divmod* :: *uint8* \Rightarrow *uint8* \Rightarrow *uint8* \times *uint8* **where**
uint8-divmod *x y* =
 (if *y* = 0 then (undefined ((*div*) :: *uint8* \Rightarrow -) *x* (0 :: *uint8*), undefined ((*mod*) ::
uint8 \Rightarrow -) *x* (0 :: *uint8*))
 else (*x div y*, *x mod y*)

definition *uint8-div* :: *uint8* \Rightarrow *uint8* \Rightarrow *uint8*
where *uint8-div* *x y* = *fst* (*uint8-divmod* *x y*)

definition *uint8-mod* :: *uint8* \Rightarrow *uint8* \Rightarrow *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* **unfolding** *uint8-divmod-def* *uint8-div-def*
by *transfer* (*simp* *add*: *word-div-def*)

lemma *mod-uint8-code* [*code*]: *x mod y* = (if *y* = 0 then *x* else *uint8-mod* *x y*)
including *undefined-transfer* **unfolding** *uint8-mod-def* *uint8-divmod-def*
by *transfer* (*simp* *add*: *word-mod-def*)

definition *uint8-sdiv* :: *uint8* \Rightarrow *uint8* \Rightarrow *uint8*
where
uint8-sdiv *x y* =
 (if *y* = 0 then undefined ((*div*) :: *uint8* \Rightarrow -) *x* (0 :: *uint8*)
 else *Abs-uint8* (*Rep-uint8* *x sdiv Rep-uint8* *y*)

definition *div0-uint8* :: *uint8* \Rightarrow *uint8*
where [*code del*]: *div0-uint8* *x* = undefined ((*div*) :: *uint8* \Rightarrow -) *x* (0 :: *uint8*)
declare [[*code abort*: *div0-uint8*]]

definition *mod0-uint8* :: *uint8* \Rightarrow *uint8*
where [*code del*]: *mod0-uint8* *x* = undefined ((*mod*) :: *uint8* \Rightarrow -) *x* (0 :: *uint8*)
declare [[*code abort*: *mod0-uint8*]]

lemma *uint8-divmod-code* [*code*]:
uint8-divmod *x y* =
 (if $0x80 \leq y$ then if $x < y$ then (0, *x*) else (1, *x* - *y*)
 else if *y* = 0 then (*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* **unfolding** *uint8-divmod-def* *uint8-sdiv-def* *div0-uint8-def*
mod0-uint8-def
less-eq-uint8.rep-eq
apply *transfer*
apply (*simp* *add*: *divmod-via-sdivmod* *push-bit-eq-mult*)

done

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)

unfolding *uint8-sdiv-def* **by**(*simp add: Abs-uint8-inverse*)

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

definition *uint8-test-bit* :: *uint8* \Rightarrow *integer* \Rightarrow *bool*

where [code del]:

uint8-test-bit x n =
(if n < 0 ∨ 7 < n then undefined (bit :: uint8 => -) x n
else bit x (nat-of-integer n))

lemma *bit-uint8-code* [code]:

bit x n \longleftrightarrow n < 8 ∧ uint8-test-bit x (integer-of-nat n)
including *undefined-transfer integer.lifting* **unfolding** *uint8-test-bit-def*
by (*transfer, simp, transfer, simp*)

lemma *uint8-test-bit-code* [code]:

uint8-test-bit w n =
(if n < 0 ∨ 7 < n then undefined (bit :: uint8 => -) w n else bit (Rep-uint8 w)
(nat-of-integer n))
unfolding *uint8-test-bit-def*
by (*simp add: bit-uint8.rep-eq*)

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

(SML) Uint8.test'-bit and
(Haskell) Data'-Bits.testBitBounded and
(Scala) Uint8.test'-bit and
(Eval) (fn x => fn i => if i < 0 orelse i >= 8 then raise (Fail argument to

uint8'-test'-bit out of bounds) else Uint8.test'-bit x i)

definition *uint8-set-bit* :: *uint8* \Rightarrow *integer* \Rightarrow *bool* \Rightarrow *uint8*

where [*code del*]:

uint8-set-bit x n b =

(*if n < 0* \vee *7 < n* then *undefined* (*set-bit* :: *uint8* \Rightarrow -) *x n b*
else *set-bit x (nat-of-integer n) b*)

lemma *set-bit-uint8-code* [*code*]:

set-bit x n b = (*if n < 8* then *uint8-set-bit x (integer-of-nat n) b* else *x*)

including *undefined-transfer integer.lifting unfolding uint8-set-bit-def*

by(*transfer*)(*auto cong; conj-cong simp add: not-less set-bit-beyond word-size*)

lemma *uint8-set-bit-code* [*code*]:

Rep-uint8 (uint8-set-bit w n b) =

(*if n < 0* \vee *7 < n* then *Rep-uint8 (undefined (set-bit :: uint8 \Rightarrow -) w n b)*
else *set-bit (Rep-uint8 w) (nat-of-integer n) b*)

including *undefined-transfer unfolding uint8-set-bit-def* **by** *transfer simp*

code-printing constant *uint8-set-bit* \mapsto

(*SML*) *Uint8.set'-bit* **and**

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

(*Scala*) *Uint8.set'-bit* **and**

(*Eval*) (*fn x => fn i => fn b => if i < 0 orelse i >= 8 then raise (Fail argument to uint8'-set'-bit out of bounds) else Uint8.set'-bit x i b*)

definition *uint8-shiffl* :: *uint8* \Rightarrow *integer* \Rightarrow *uint8*

where [*code del*]:

uint8-shiffl x n = (*if n < 0* \vee *8 \leq n* then *undefined (push-bit :: nat \Rightarrow uint8 \Rightarrow -) x n* else *push-bit (nat-of-integer n) x*)

lemma *shiffl-uint8-code* [*code*]:

push-bit n x = (*if n < 8* then *uint8-shiffl x (integer-of-nat n)* else *0*)

including *undefined-transfer integer.lifting unfolding uint8-shiffl-def*

by *transfer simp*

lemma *uint8-shiffl-code* [*code*]:

Rep-uint8 (uint8-shiffl w n) =

(*if n < 0* \vee *8 \leq n* then *Rep-uint8 (undefined (push-bit :: nat \Rightarrow uint8 \Rightarrow -) w n)*
else *push-bit (nat-of-integer n) (Rep-uint8 w)*)

including *undefined-transfer unfolding uint8-shiffl-def*

by *transfer simp*

code-printing constant *uint8-shiffl* \mapsto

(*SML*) *Uint8.shiffl* **and**

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

(*Scala*) *Uint8.shiffl* **and**

(Eval) (fn x => fn i => if i < 0 orelse i >= 8 then raise (Fail argument to uint8'-shiftr out of bounds) else Uint8.shiftr x i)

definition *uint8-shiftr* :: *uint8* ⇒ *integer* ⇒ *uint8*

where [code del]:

uint8-shiftr x n = (if n < 0 ∨ 8 ≤ n then undefined (drop-bit :: - ⇒ - ⇒ uint8) x n else drop-bit (nat-of-integer n) x)

lemma *shiftr-uint8-code* [code]:

drop-bit n x = (if n < 8 then *uint8-shiftr* x (integer-of-nat n) else 0)

including *undefined-transfer integer.lifting unfolding uint8-shiftr-def*

by *transfer simp*

lemma *uint8-shiftr-code* [code]:

Rep-uint8 (*uint8-shiftr* w n) =

(if n < 0 ∨ 8 ≤ n then *Rep-uint8* (undefined (drop-bit :: - ⇒ - ⇒ uint8) w n)

else *drop-bit* (nat-of-integer n) (*Rep-uint8* w))

including *undefined-transfer unfolding uint8-shiftr-def* **by** *transfer simp*

code-printing constant *uint8-shiftr* ↪

(SML) *Uint8.shiftr* **and**

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

(Scala) *Uint8.shiftr* **and**

(Eval) (fn x => fn i => if i < 0 orelse i >= 8 then raise (Fail argument to uint8'-shiftr out of bounds) else *Uint8.shiftr* x i)

definition *uint8-sshiftr* :: *uint8* ⇒ *integer* ⇒ *uint8*

where [code del]:

uint8-sshiftr x n =

(if n < 0 ∨ 8 ≤ n then undefined *signed-drop-bit-uint8* n x else *signed-drop-bit-uint8* (nat-of-integer n) x)

lemma *sshiftr-uint8-code* [code]:

signed-drop-bit-uint8 n x =

(if n < 8 then *uint8-sshiftr* x (integer-of-nat n) else if bit x 7 then -1 else 0)

including *undefined-transfer integer.lifting unfolding uint8-sshiftr-def*

by *transfer (simp add: not-less signed-drop-bit-beyond word-size)*

lemma *uint8-sshiftr-code* [code]:

Rep-uint8 (*uint8-sshiftr* w n) =

(if n < 0 ∨ 8 ≤ n then *Rep-uint8* (undefined *signed-drop-bit-uint8* n w)

else *signed-drop-bit* (nat-of-integer n) (*Rep-uint8* w))

including *undefined-transfer unfolding uint8-sshiftr-def*

by *transfer simp*

code-printing constant *uint8-sshiftr* ↪

(SML) *Uint8.shiftr'-signed* **and**

(Haskell)

(*Prelude.fromInteger* (*Prelude.toInteger* (*Data'-Bits.shiftrBounded* (*Prelude.fromInteger*

(*Prelude.toInteger* -) :: *Uint8.Int8* -) :: *Uint8.Word8*) **and**
 (*Scala*) *Uint8.shiftr'-signed* **and**
 (*Eval*) (fn x => fn i => if i < 0 orelse i >= 8 then raise (Fail argument to *uint8'-sshiftr* out of bounds) else *Uint8.shiftr'-signed* x i)

context

includes *bit-operations-syntax*

begin

lemma *uint8-msb-test-bit*: $msb\ x \longleftrightarrow bit\ (x :: uint8)\ 7$
by *transfer* (*simp* *add*: *msb-word-iff-bit*)

lemma *msb-uint16-code* [*code*]: $msb\ x \longleftrightarrow uint8\text{-test-bit}\ x\ 7$
by (*simp* *add*: *uint8-test-bit-def* *uint8-msb-test-bit*)

lemma *uint8-of-int-code* [*code*]:
uint8-of-int i = *Uint8* (*integer-of-int* i)
including *integer.lifting* **by** *transfer* *simp*

lemma *int-of-uint8-code* [*code*]:
int-of-uint8 x = *int-of-integer* (*integer-of-uint8* x)
by (*simp* *add*: *int-of-uint8.rep-eq* *integer-of-uint8-def*)

lemma *uint8-of-nat-code* [*code*]:
uint8-of-nat = *uint8-of-int* \circ *int*
by *transfer* (*simp* *add*: *fun-eq-iff*)

lemma *nat-of-uint8-code* [*code*]:
nat-of-uint8 x = *nat-of-integer* (*integer-of-uint8* x)
unfolding *integer-of-uint8-def* **including** *integer.lifting* **by** *transfer* *simp*

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-uint8-signed-code* [*code*]:
integer-of-uint8-signed n =
 (if bit n 7 then undefined *integer-of-uint8* n else *integer-of-int* (*uint* (*Rep-uint8'*
 n))))
by (*simp* *add*: *integer-of-uint8-signed-def* *integer-of-uint8-def*)

lemma *integer-of-uint8-code* [*code*]:
integer-of-uint8 n =
 (if bit n 7 then *integer-of-uint8-signed* (n AND 0x7F) OR 0x80 else *integer-of-uint8-signed*
 n)

proof —

have $\langle integer-of-uint8-signed\ (n\ AND\ 0x7F)\ OR\ 0x80 = Bit-Operations.set-bit\ 7\ (integer-of-uint8-signed\ (take-bit\ 7\ n)) \rangle$


```

  by (simp add: take-bit-eq-mask set-bit-eq-or push-bit-eq-mult mask-eq-exp-minus-1)
  moreover have ⟨integer-of-uint8 n = Bit-Operations.set-bit 7 (integer-of-uint8
(take-bit 7 n))⟩ if ⟨bit n 7⟩
  proof (rule bit-eqI)
    fix m
    from that show ⟨bit (integer-of-uint8 n) m = bit (Bit-Operations.set-bit 7
(integer-of-uint8 (take-bit 7 n))) m⟩ for m
    including integer.lifting by transfer (auto simp add: bit-simps dest: bit-imp-le-length)
  qed
  ultimately show ?thesis
  by simp (simp add: integer-of-uint8-signed-def bit-simps)
qed

```

end

code-printing

```

  constant integer-of-uint8 →
  (SML) IntInf.fromLarge (Word8.toLargeInt -) and
  (Haskell) Prelude.toInteger
| constant integer-of-uint8-signed →
  (Scala) BigInt

```

9.3 Quickcheck setup

definition *uint8-of-natural* :: *natural* ⇒ *uint8*
where *uint8-of-natural* *x* ≡ *Uint8* (*integer-of-natural* *x*)

instantiation *uint8* :: {*random*, *exhaustive*, *full-exhaustive*} **begin**

definition *random-uint8* ≡ *qc-random-cnv uint8-of-natural*

definition *exhaustive-uint8* ≡ *qc-exhaustive-cnv uint8-of-natural*

definition *full-exhaustive-uint8* ≡ *qc-full-exhaustive-cnv uint8-of-natural*

instance ..

end

instantiation *uint8* :: *narrowing* **begin**

interpretation *quickcheck-narrowing-samples*

λi. let x = Uint8 i in (x, 0xFF - x) 0

Typerep.Typerep (STR "Uint8.uint8") [] .

definition *narrowing-uint8* *d* = *qc-narrowing-drawn-from* (*narrowing-samples* *d*)
d

declare [[*code drop: partial-term-of :: uint8 itself ⇒ -*]]

lemmas *partial-term-of-uint8* [*code*] = *partial-term-of-code*

instance ..

end

end

Chapter 10

Unsigned words of default size

theory *Uint* **imports**

Code-Target-Word-Base Word-Type-Copies

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* $\equiv \lambda - :: dflt-size\ itself. Typerep.Type\ rep\ (STR\ "Uint.dflt-size")$ \square

instance ..

end

consts *dflt-size-aux* :: *nat*

specification (*dflt-size-aux*) *dflt-size-aux-g0*: *dflt-size-aux* > 0

by *auto*

hide-fact *dflt-size-aux-def*

instantiation *dflt-size* :: *len* **begin**

definition *len-of-dflt-size* (*-* :: *dflt-size* *itself*) $\equiv dflt-size-aux$

instance **by**(*intro-classes*)(*simp* *add*: *len-of-dflt-size-def* *dflt-size-aux-g0*)

end

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

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

lift-definition $dflt\text{-}size\text{-}integer :: integer \text{ is } int \text{ dflt-size} .$

declare $dflt\text{-}size\text{-}integer\text{-}del [code del]$

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

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

by *transfer simp*

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

$dflt\text{-}size > 0$

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

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

using $len\text{-}gt\text{-}0 [where 'a = dflt-size]$

by (*simp-all del: len-gt-0*)

end

10.1 Type definition and primitive operations

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

global-interpretation $uint: word\text{-}type\text{-}copy \text{ Abs-uint } Rep\text{-}uint$

using $type\text{-}definition\text{-}uint \text{ by } (rule \text{ word-type-copy.intro})$

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

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

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

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

begin

lift-definition $zero\text{-}uint :: uint \text{ is } 0 .$

lift-definition $one\text{-}uint :: uint \text{ is } 1 .$

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

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

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

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

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

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

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

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

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

global-interpretation $uint: word\text{-}type\text{-}copy\text{-}ring \text{ Abs-uint } Rep\text{-}uint$

by *standard (fact zero-uint.rep-eq one-uint.rep-eq)*

*plus-uint.rep-eq uminus-uint.rep-eq minus-uint.rep-eq
times-uint.rep-eq divide-uint.rep-eq modulo-uint.rep-eq
equal-uint.rep-eq less-eq-uint.rep-eq less-uint.rep-eq)+*

instance proof –

show $\langle OFCLASS(uint, comm-ring-1-class) \rangle$
by (*rule uint.of-class-comm-ring-1*)
show $\langle OFCLASS(uint, semiring-modulo-class) \rangle$
by (*fact uint.of-class-semiring-modulo*)
show $\langle OFCLASS(uint, equal-class) \rangle$
by (*fact uint.of-class-equal*)
show $\langle OFCLASS(uint, linorder-class) \rangle$
by (*fact uint.of-class-linorder*)

qed

end

instantiation *uint :: ring-bit-operations*

begin

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

global-interpretation *uint: word-type-copy-bits Abs-uint Rep-uint signed-drop-bit-uint*

by *standard* (*fact bit-uint.rep-eq not-uint.rep-eq and-uint.rep-eq or-uint.rep-eq
xor-uint.rep-eq*

*mask-uint.rep-eq push-bit-uint.rep-eq drop-bit-uint.rep-eq signed-drop-bit-uint.rep-eq
take-bit-uint.rep-eq*

set-bit-uint.rep-eq unset-bit-uint.rep-eq flip-bit-uint.rep-eq)+

instance

by (*fact uint.of-class-ring-bit-operations*)

end

lift-definition *uint-of-nat* :: $\langle nat \Rightarrow uint \rangle$

is *word-of-nat* .

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

lift-definition *wint-of-int* :: $\langle \text{int} \Rightarrow \text{uint} \rangle$
is *word-of-int* .

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

context
includes *integer.lifting*
begin

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

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

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
apply *standard*
apply (*simp-all add: uint-of-nat.rep-eq nat-of-uint.rep-eq*
uint-of-int.rep-eq int-of-uint.rep-eq
Uint.rep-eq integer-of-uint.rep-eq integer-eq-iff)
done

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

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

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

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

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

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

context
includes *lifting-syntax*
begin

lemma *set-bit-uint-transfer* [*transfer-rule*]:
 $\langle (\text{cr-uint} \text{ ===} \rangle (=) \text{ ===} \rangle (\longleftrightarrow) \text{ ===} \rangle \text{cr-uint} \rangle$ *Generic-set-bit.set-bit Generic-set-bit.set-bit*
by (*simp only: set-bit-eq [abs-def] set-bit-uint-eq [abs-def]*) *transfer-prover*

end

lift-definition *set-bits-uint* :: $\langle (nat \Rightarrow bool) \Rightarrow uint \rangle$ **is** *set-bits* .

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

.

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
by (*standard*; *transfer*) *simp-all*

instance using *uint.of-class-bit-comprehension*

uint.of-class-set-bit uint.of-class-lsb

by *simp-all standard*

end

10.2 Code setup

code-printing code-module *Uint* \rightarrow (*SML*)

\langle

structure Uint : *sig*

val set-bit : *Word.word* \rightarrow *IntInf.int* \rightarrow *bool* \rightarrow *Word.word*

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 set-bit *x n b* =

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

in if b then Word.orb (*x*, *mask*)

else Word.andb (*x*, *Word.notb mask*)

end

fun shiftl *x n* =

Word.<< (*x*, *Word.fromLargeInt* (*IntInf.toLarge n*))

fun shiftr *x n* =

Word.>> (*x*, *Word.fromLargeInt* (*IntInf.toLarge n*))

fun shiftr-signed *x n* =

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

fun test-bit *x n* =

Word.andb (*x*, *Word.<<* (*0wx1*, *Word.fromLargeInt* (*IntInf.toLarge n*))) $\langle \rangle$

Word.fromInt 0

end; (* *struct Uint* *) \rangle

code-reserved *SML Uint*

code-printing code-module *Uint* \rightarrow (*Haskell*)

\langle 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*)

\langle 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
```

\rangle

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*)

\langle module *Uint* : sig

```
type t = int
val dflt-size : Z.t
val less : t -> t -> bool
val less-eq : t -> t -> bool
val set-bit : t -> Z.t -> bool -> t
val shiftl : t -> Z.t -> t
val shiftr : t -> Z.t -> t
val shiftr-signed : t -> Z.t -> t
val test-bit : t -> Z.t -> bool
val int-mask : int
val int32-mask : int32
val int64-mask : int64
end = struct
```

type t = int


```

let dftt-size = Z.of-int Sys.int-size;;

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

let less-eq x y =
  if x < 0 then
    y < 0 && x <= y
  else y < 0 || x <= y;;

let set-bit x n b =
  let mask = 1 lsl (Z.to-int n)
  in if b then x lor mask
     else x land (lnot mask);;

let shiffl 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 dftt-size : BigInt = BigInt(32)

def less(x: Int, y: Int) : Boolean =
  if (x < 0) y < 0 && x < y
  else y < 0 || x < y

```

```

def less-eq(x: Int, y: Int) : Boolean =
  if (x < 0) y < 0 && x <= y
  else y < 0 || x <= y

def set-bit(x: Int, n: BigInt, b: Boolean) : Int =
  if (b)
    x | (1 << n.intValue)
  else
    x & (1 << n.intValue).unary~

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 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* .

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

by *transfer (simp add: wivs-mask-def)*

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

lift-definition *wivs-shift-integer* :: *integer* **is** *wivs-shift* .

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

by *transfer (simp add: wivs-shift-def)*

definition *wivs-index* :: *nat* **where** *wivs-index* == *dflt-size* - 1

lift-definition *wivs-index-integer* :: *integer* **is** *int wivs-index* .

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

by *transfer (simp add: wivs-index-def of-nat-diff)*

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

lift-definition *wivs-overflow-integer* :: *integer* **is** *wivs-overflow* .

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

by *transfer (simp add: wivs-overflow-def)*

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

lift-definition *wivs-least-integer* :: integer is *wivs-least* .
lemma [code]: *wivs-least-integer* = $-(2^{(dflt-size - 1)})$
by transfer (simp add: *wivs-overflow-def wivs-least-def*)

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*
unfolding *Uint-signed-def*
apply transfer
apply (subst *word-of-int-via-signed*)
apply (auto simp add: *push-bit-of-1 mask-eq-exp-minus-1 word-of-int-via-signed wivs-mask-def wivs-index-def wivs-overflow-def wivs-least-def wivs-shift-def*)
done

lemma *Uint-signed-code* [code]:
Rep-uint (*Uint-signed* *i*) =
 (if *i* < *wivs-least-integer* \vee *i* \geq *wivs-overflow-integer* then *Rep-uint* (undefined *Uint* *i*) else *word-of-int* (*int-of-integer-symbolic* *i*))
unfolding *Uint-signed-def Uint-def int-of-integer-symbolic-def word-of-integer-def*
by(simp add: *Abs-uint-inverse*)
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*)
unfolding *Rep-uint'-def* **by** transfer (simp add: *set-bits-bit-eq*)

lift-definition *Abs-uint'* :: *dflt-size* word \Rightarrow uint is $\lambda x ::$ *dflt-size* word. *x* .

lemma *Abs-uint'-code* [code]:
Abs-uint' *x* = *Uint* (*integer-of-int* (*uint* *x*))
including *integer.lifting* **by** transfer simp

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

```

lemma term-of-uint-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 "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))
  by(simp add: term-of-anything)

```

Important: We must prevent the reflection oracle (*eval-tac*) to use our machine-dependent type.

code-printing

```

type-constructor uint  $\rightarrow$ 
  (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  $\rightarrow$ 
  (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  $\rightarrow$ 
  (SML) Word.fromLargeInt (IntInf.toLarge -) and
  (Eval) (raise (Fail Machine dependent code)) and
  (Quickcheck) Word.fromInt and
  (Haskell) (Prelude.fromInteger - :: Uint.Word) and
  (Haskell-Quickcheck) (Prelude.fromInteger (Prelude.toInteger -) :: Uint.Word)
and
  (Scala) -.intValue
| constant Uint-signed  $\rightarrow$ 
  (OCaml) Z.to'-int
| constant 0 :: uint  $\rightarrow$ 
  (SML) (Word.fromInt 0) and
  (Eval) (raise (Fail Machine dependent code)) and
  (Quickcheck) (Word.fromInt 0) and
  (Haskell) (0 :: Uint.Word) and
  (OCaml) 0 and
  (Scala) 0
| constant 1 :: uint  $\rightarrow$ 
  (SML) (Word.fromInt 1) and
  (Eval) (raise (Fail Machine dependent code)) and
  (Quickcheck) (Word.fromInt 1) and

```

```

(Haskell) (1 :: Uint.Word) and
(OCaml) 1 and
(Scala) 1
| constant plus :: uint ⇒ - →
(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 $uint-divmod :: uint \Rightarrow uint \Rightarrow uint \times uint$ **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 $uint-div :: uint \Rightarrow uint \Rightarrow uint$
where $uint-div\ x\ y = fst\ (uint-divmod\ x\ y)$

definition $uint-mod :: uint \Rightarrow uint \Rightarrow uint$
where $uint-mod\ x\ y = snd\ (uint-divmod\ x\ y)$

lemma $div-uint-code\ [code]:\ x\ div\ y = (if\ y = 0\ then\ 0\ else\ uint-div\ x\ y)$
including $undefined-transfer$ **unfolding** $uint-divmod-def\ uint-div-def$
by $transfer(simp\ add:\ word-div-def)$

lemma *mod-uint-code* [code]: $x \bmod y = (\text{if } y = 0 \text{ then } x \text{ else } \text{uint-mod } x \ y)$
including *undefined-transfer unfolding uint-mod-def uint-divmod-def*
by *transfer(simp add: word-mod-def)*

definition *uint-sdiv* :: $\text{uint} \Rightarrow \text{uint} \Rightarrow \text{uint}$
where [code del]:
uint-sdiv $x \ y =$
 ($\text{if } y = 0 \text{ then undefined } ((\text{div}) :: \text{uint} \Rightarrow -) \ x \ (0 :: \text{uint})$
 else $\text{Abs-uint } (\text{Rep-uint } x \ \text{sdiv } \text{Rep-uint } y)$)

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

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

definition *wivs-overflow-uint* :: uint
where $\text{wivs-overflow-uint} \equiv \text{push-bit } (\text{dflt-size} - 1) \ 1$

lemma *Rep-uint-wivs-overflow-uint-eq*:
 $\langle \text{Rep-uint } \text{wivs-overflow-uint} = 2^{\wedge} (\text{dflt-size} - \text{Suc } 0) \rangle$
by (*simp add: wivs-overflow-uint-def one-uint.rep-eq push-bit-uint.rep-eq uint.word-of-power push-bit-eq-mult*)

lemma *wivs-overflow-uint-greater-eq-0*:
 $\langle \text{wivs-overflow-uint} > 0 \rangle$
apply (*simp add: less-uint.rep-eq zero-uint.rep-eq Rep-uint-wivs-overflow-uint-eq*)
apply *transfer*
apply (*simp add: take-bit-push-bit push-bit-eq-mult*)
done

lemma *uint-divmod-code* [code]:
uint-divmod $x \ y =$
 ($\text{if } \text{wivs-overflow-uint} \leq y \text{ then if } x < y \text{ then } (0, x) \text{ else } (1, x - y)$
 else $\text{if } y = 0 \text{ then } (\text{div0-uint } x, \text{mod0-uint } x)$
 else $\text{let } q = \text{push-bit } 1 \ (\text{uint-sdiv } (\text{drop-bit } 1 \ x) \ y);$
 $r = x - q * y$
 $\text{in if } r \geq y \text{ then } (q + 1, r - y) \text{ else } (q, r)$)

proof (*cases* $\langle y = 0 \rangle$)
case *True*
moreover have $\langle x \geq 0 \rangle$
by *transfer simp*
moreover note *wivs-overflow-uint-greater-eq-0*
ultimately show *?thesis*
by (*auto simp add: uint-divmod-def div0-uint-def mod0-uint-def not-less*)
next

```

case False
then show ?thesis
  including undefined-transfer
  unfolding uint-divmod-def uint-sdiv-def div0-uint-def mod0-uint-def
    wivs-overflow-uint-def
  apply transfer
  apply (simp add: divmod-via-sdivmod push-bit-of-1)
  done
qed

```

```

lemma uint-sdiv-code [code]:
  Rep-uint (uint-sdiv x y) =
    (if y = 0 then Rep-uint (undefined ((div) :: uint ⇒ -) x (0 :: uint))
     else Rep-uint x sdiv Rep-uint y)
unfolding uint-sdiv-def by(simp add: Abs-uint-inverse)

```

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

code-printing

```

constant uint-div  $\rightarrow$ 
  (SML) Word.div ((-), (-)) and
  (Eval) (raise (Fail Machine dependent code)) and
  (Quickcheck) Word.div ((-), (-)) and
  (Haskell) Prelude.div
| constant uint-mod  $\rightarrow$ 
  (SML) Word.mod ((-), (-)) and
  (Eval) (raise (Fail Machine dependent code)) and
  (Quickcheck) Word.mod ((-), (-)) and
  (Haskell) Prelude.mod
| constant uint-divmod  $\rightarrow$ 
  (Haskell) divmod
| constant uint-sdiv  $\rightarrow$ 
  (OCaml) Pervasives.(/) and
  (Scala) - '/' -

```

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

where [*code del*]:

```

uint-test-bit x n =
  (if n < 0 ∨ dflt-size-integer ≤ n then undefined (bit :: uint ⇒ -) x n
   else bit x (nat-of-integer n))

```

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

```

bit x n  $\longleftrightarrow$  n < dflt-size ∧ uint-test-bit x (integer-of-nat n)
including undefined-transfer integer.lifting unfolding uint-test-bit-def
by (transfer, simp, transfer, simp)

```


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

uint-test-bit w n =
 (if $n < 0 \vee \text{dflt-size-integer} \leq n$ then undefined (*bit* :: *uint* \Rightarrow -) w n else *bit*
 (*Rep-uint* w) (*nat-of-integer* n))
unfolding *uint-test-bit-def* **by** (*simp add: bit-uint.rep-eq*)

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

(*SML*) *Uint.test'-bit* **and**
 (*Eval*) (*raise (Fail Machine dependent code)*) **and**
 (*Quickcheck*) *Uint.test'-bit* **and**
 (*Haskell*) *Data'-Bits.testBitBounded* **and**
 (*OCaml*) *Uint.test'-bit* **and**
 (*Scala*) *Uint.test'-bit*

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

where [code del]:

uint-set-bit x n b =
 (if $n < 0 \vee \text{dflt-size-integer} \leq n$ then undefined (*set-bit* :: *uint* \Rightarrow -) x n b
 else *set-bit* x (*nat-of-integer* n) b)

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

set-bit x n b = (if $n < \text{dflt-size}$ then *uint-set-bit* x (*integer-of-nat* n) b else x)
including *undefined-transfer integer.lifting* **unfolding** *uint-set-bit-def*
by (*transfer*) (*auto cong: conj-cong simp add: not-less set-bit-beyond word-size*)

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

Rep-uint (*uint-set-bit* w n b) =
 (if $n < 0 \vee \text{dflt-size-integer} \leq n$ then *Rep-uint* (undefined (*set-bit* :: *uint* \Rightarrow -) w
 n b)
 else *set-bit* (*Rep-uint* w) (*nat-of-integer* n) b)

including *undefined-transfer integer.lifting* **unfolding** *uint-set-bit-def* **by** *transfer simp*

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

(*SML*) *Uint.set'-bit* **and**
 (*Eval*) (*raise (Fail Machine dependent code)*) **and**
 (*Quickcheck*) *Uint.set'-bit* **and**
 (*Haskell*) *Data'-Bits.setBitBounded* **and**
 (*OCaml*) *Uint.set'-bit* **and**
 (*Scala*) *Uint.set'-bit*

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

where [code del]:

uint-shiffl x n = (if $n < 0 \vee \text{dflt-size-integer} \leq n$ then undefined (*push-bit* :: *nat*
 \Rightarrow *uint* \Rightarrow -) x n else *push-bit* (*nat-of-integer* n) x)

lemma *shiffl-uint-code* [code]: *push-bit* n x = (if $n < \text{dflt-size}$ then *uint-shiffl* x
 (*integer-of-nat* n) else 0)

including *undefined-transfer integer.lifting unfolding uint-shiffl-def*
by (*transfer fixing: n simp*)

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

Rep-uint (uint-shiffl w n) =
(if n < 0 ∨ dflt-size-integer ≤ n then Rep-uint (undefined (push-bit :: nat ⇒ uint
 \Rightarrow $-$) *w n) else push-bit (nat-of-integer n) (Rep-uint w))*
including *undefined-transfer integer.lifting unfolding uint-shiffl-def* **by** *transfer simp*

code-printing constant *uint-shiffl* \rightarrow

(*SML*) *Uint.shiffl* **and**
(*Eval*) (*raise (Fail Machine dependent code)*) **and**
(*Quickcheck*) *Uint.shiffl* **and**
(*Haskell*) *Data'-Bits.shifflBounded* **and**
(*OCaml*) *Uint.shiffl* **and**
(*Scala*) *Uint.shiffl*

definition *uint-shiftr* :: *uint ⇒ integer ⇒ uint*

where [code del]:

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

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

including *undefined-transfer integer.lifting unfolding uint-shiftr-def*
by (*transfer fixing: n simp*)

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

Rep-uint (uint-shiftr w n) =
(if n < 0 ∨ dflt-size-integer ≤ n then Rep-uint (undefined (drop-bit :: nat ⇒ uint
 \Rightarrow $-$) *w n) else drop-bit (nat-of-integer n) (Rep-uint w))*
including *undefined-transfer integer.lifting unfolding uint-shiftr-def* **by** *transfer simp*

code-printing constant *uint-shiftr* \rightarrow

(*SML*) *Uint.shiftr* **and**
(*Eval*) (*raise (Fail Machine dependent code)*) **and**
(*Quickcheck*) *Uint.shiftr* **and**
(*Haskell*) *Data'-Bits.shiftrBounded* **and**
(*OCaml*) *Uint.shiftr* **and**
(*Scala*) *Uint.shiftr*

definition *uint-sshiftr* :: *uint ⇒ integer ⇒ uint*

where [code del]:

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

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

signed-drop-bit-uint n x =
(if n < dflt-size then uint-sshiftr x (integer-of-nat n) else
if bit x wivs-index then -1 else 0)

including *undefined-transfer integer.lifting unfolding uint-sshiftr-def*
by *transfer(simp add: not-less signed-drop-bit-beyond word-size wivs-index-def)*

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

Rep-uint (uint-sshiftr w n) =
(if n < 0 ∨ dflt-size-integer ≤ n then Rep-uint (undefined signed-drop-bit-uint n
w) else signed-drop-bit (nat-of-integer n) (Rep-uint w))
including *undefined-transfer integer.lifting unfolding uint-sshiftr-def* **by** *transfer*
simp

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*
by *transfer (simp add: msb-word-iff-bit wivs-index-def)*

lemma *msb-uint-code* [code]: *msb x \longleftrightarrow uint-test-bit x wivs-index-integer*
apply(*simp add: uint-test-bit-def uint-msb-test-bit*
wivs-index-integer-code dflt-size-integer-def wivs-index-def)
by (*metis (full-types) One-nat-def dflt-size(2) less-iff-diff-less-0*
nat-of-integer-of-nat of-nat-1 of-nat-diff of-nat-less-0-iff wivs-index-def)

lemma *uint-of-int-code* [code]: *uint-of-int i = (BITS n. bit i n)*
by *transfer (simp add: word-of-int-conv-set-bits)*

10.3 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 ..

end

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

interpretation *quickcheck-narrowing-samples*

$\lambda i. (Uint\ i, Uint\ (-\ i))\ 0$
Typerep.Typerep (STR "Uint.uint") [] .

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

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

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

instance ..

end

end

Chapter 11

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  
  by (simp add: integer-of-char-def char-of-integer-def)
```

```
lemma char-of-integer-integer-of-char [simp]:  
  char-of-integer (integer-of-char x) = x  
  by (simp add: integer-of-char-def char-of-integer-def)
```

```
lemma int-lt-numeral [simp]: int x < numeral n  $\longleftrightarrow$  x < numeral n  
by (metis nat-numeral zless-nat-eq-int-zless)
```

```
lemma int-of-integer-ge-0: 0  $\leq$  int-of-integer x  $\longleftrightarrow$  0  $\leq$  x  
including integer.lifting by transfer simp
```

```
lemma integer-of-char-ge-0 [simp]: 0  $\leq$  integer-of-char x  
including integer.lifting unfolding integer-of-char-def  
by transfer (simp add: of-char-def)
```

11.1 Conversion between native words

lift-definition *wint8-of-wint16* :: *wint16* ⇒ *wint8* is *ucast* .

lift-definition *wint8-of-wint32* :: *wint32* ⇒ *wint8* is *ucast* .

lift-definition *wint8-of-wint64* :: *wint64* ⇒ *wint8* is *ucast* .

lift-definition *wint16-of-wint8* :: *wint8* ⇒ *wint16* is *ucast* .

lift-definition *wint16-of-wint32* :: *wint32* ⇒ *wint16* is *ucast* .

lift-definition *wint16-of-wint64* :: *wint64* ⇒ *wint16* is *ucast* .

lift-definition *wint32-of-wint8* :: *wint8* ⇒ *wint32* is *ucast* .

lift-definition *wint32-of-wint16* :: *wint16* ⇒ *wint32* is *ucast* .

lift-definition *wint32-of-wint64* :: *wint64* ⇒ *wint32* is *ucast* .

lift-definition *wint64-of-wint8* :: *wint8* ⇒ *wint64* is *ucast* .

lift-definition *wint64-of-wint16* :: *wint16* ⇒ *wint64* is *ucast* .

lift-definition *wint64-of-wint32* :: *wint32* ⇒ *wint64* is *ucast* .

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 uint32-of-uint8  $\rightarrow$ 
(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))
by transfer simp

```

```

lemma uint8-of-uint32-code [code]:
  uint8-of-uint32 x = Abs-uint8' (ucast (Rep-uint32' x))
by transfer simp

```

```

lemma uint8-of-uint64-code [code]:
  uint8-of-uint64 x = Abs-uint8' (ucast (Rep-uint64' x))
by transfer simp

```

```

lemma uint16-of-uint8-code [code]:
  uint16-of-uint8 x = Abs-uint16' (ucast (Rep-uint8' x))
by transfer simp

```

```

lemma uint16-of-uint32-code [code]:

```

$uint16\text{-of-}uint32\ x = Abs\text{-}uint16'\ (ucast\ (Rep\text{-}uint32'\ x))$
by *transfer simp*

lemma *uint16-of-uint64-code* [code]:
 $uint16\text{-of-}uint64\ x = Abs\text{-}uint16'\ (ucast\ (Rep\text{-}uint64'\ x))$
by *transfer simp*

lemma *uint32-of-uint8-code* [code]:
 $uint32\text{-of-}uint8\ x = Abs\text{-}uint32'\ (ucast\ (Rep\text{-}uint8'\ x))$
by *transfer simp*

lemma *uint32-of-uint16-code* [code]:
 $uint32\text{-of-}uint16\ x = Abs\text{-}uint32'\ (ucast\ (Rep\text{-}uint16'\ x))$
by *transfer simp*

lemma *uint32-of-uint64-code* [code]:
 $uint32\text{-of-}uint64\ x = Abs\text{-}uint32'\ (ucast\ (Rep\text{-}uint64'\ x))$
by *transfer simp*

lemma *uint64-of-uint8-code* [code]:
 $uint64\text{-of-}uint8\ x = Abs\text{-}uint64'\ (ucast\ (Rep\text{-}uint8'\ x))$
by *transfer simp*

lemma *uint64-of-uint16-code* [code]:
 $uint64\text{-of-}uint16\ x = Abs\text{-}uint64'\ (ucast\ (Rep\text{-}uint16'\ x))$
by *transfer simp*

lemma *uint64-of-uint32-code* [code]:
 $uint64\text{-of-}uint32\ x = Abs\text{-}uint64'\ (ucast\ (Rep\text{-}uint32'\ x))$
by *transfer simp*

end

theory *Native-Cast-Uint* **imports**

Native-Cast

Uint

begin

lift-definition *uint-of-uint8* :: $uint8 \Rightarrow uint$ **is** *ucast* .

lift-definition *uint-of-uint16* :: $uint16 \Rightarrow uint$ **is** *ucast* .

lift-definition *uint-of-uint32* :: $uint32 \Rightarrow uint$ **is** *ucast* .

lift-definition *uint-of-uint64* :: $uint64 \Rightarrow uint$ **is** *ucast* .

lift-definition *uint8-of-uint* :: $uint \Rightarrow uint8$ **is** *ucast* .

lift-definition *uint16-of-uint* :: $uint \Rightarrow uint16$ **is** *ucast* .

lift-definition *uint32-of-uint* :: $uint \Rightarrow uint32$ **is** *ucast* .

lift-definition *uint64-of-uint* :: $uint \Rightarrow uint64$ **is** *ucast* .

code-printing

```

constant uint-of-uint8  $\rightarrow$ 
  (SML) Word.fromLarge (Word8.toLarge -) and
  (Haskell) (Prelude.fromIntegral - :: Uint.Word) and
  (Scala) ((-).toInt & 0xFF)
| constant uint-of-uint16  $\rightarrow$ 
  (SML-word) Word.fromLarge (Word16.toLarge -) and
  (Haskell) (Prelude.fromIntegral - :: Uint.Word) and
  (Scala) ((-).toInt)
| constant uint-of-uint32  $\rightarrow$ 
  (SML) Word.fromLarge (Word32.toLarge -) and
  (Haskell) (Prelude.fromIntegral - :: Uint.Word) and
  (Scala) - and
  (OCaml) (Int32.of'-int -) and Uint.int'-mask
| constant uint-of-uint64  $\rightarrow$ 
  (SML) Word.fromLarge (Uint64.toLarge -) and
  (Haskell) (Prelude.fromIntegral - :: Uint.Word) and
  (Scala) ((-).toInt) and
  (OCaml) Int64.of'-int
| constant uint8-of-uint  $\rightarrow$ 
  (SML) Word8.fromLarge (Word.toLarge -) and
  (Haskell) (Prelude.fromIntegral - :: Uint8.Word8) and
  (Scala) ((-).toByte)
| constant uint16-of-uint  $\rightarrow$ 
  (SML-word) Word16.fromLarge (Word.toLarge -) and
  (Haskell) (Prelude.fromIntegral - :: Uint16.Word16) and
  (Scala) ((-).toChar)
| constant uint32-of-uint  $\rightarrow$ 
  (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  $\rightarrow$ 
  (SML) Uint64.fromLarge (Word.toLarge -) and
  (Haskell) (Prelude.fromIntegral - :: Uint64.Word64) and
  (Scala) ((-).toLong & 0xFFFFFFFFL) and
  (OCaml) Int64.logand (Int64.of'-int -) Uint.int64'-mask

```

```

lemma uint8-of-uint-code [code]:
  uint8-of-uint x = Abs-uint8' (ucast (Rep-uint' x))
  unfolding Rep-uint'-def by transfer simp

```

```

lemma uint16-of-uint-code [code]:
  uint16-of-uint x = Abs-uint16' (ucast (Rep-uint' x))
  unfolding Rep-uint'-def by transfer simp

```

```

lemma uint32-of-uint-code [code]:
  uint32-of-uint x = Abs-uint32' (ucast (Rep-uint' x))
  unfolding Rep-uint'-def by transfer simp

```

```

lemma wint64-of-wint-code [code]:
  wint64-of-wint x = Abs-wint64' (ucast (Rep-wint' x))
  unfolding Rep-wint'-def by transfer simp

```

```

lemma wint-of-wint8-code [code]:
  wint-of-wint8 x = Abs-wint' (ucast (Rep-wint8' x))
  by transfer simp

```

```

lemma wint-of-wint16-code [code]:
  wint-of-wint16 x = Abs-wint' (ucast (Rep-wint16' x))
  by transfer simp

```

```

lemma wint-of-wint32-code [code]:
  wint-of-wint32 x = Abs-wint' (ucast (Rep-wint32' x))
  by transfer simp

```

```

lemma wint-of-wint64-code [code]:
  wint-of-wint64 x = Abs-wint' (ucast (Rep-wint64' x))
  by transfer simp

```

```

end

```

11.2 Compatibility with Imperative/HOL

```

theory Native-Word-Imperative-HOL imports

```

```

  Code-Target-Word-Base

```

```

  HOL-Imperative-HOL.Heap-Monad

```

```

begin

```

We add a code target that combines the translations for native words that are by default not supported by all PolyML versions with the adaptations for `Imperative_HOL`.

```

setup ‹Code-Target.add-derived-target (SML-word-imp, [(SML-word, I), (SML-imp, I)]›

```

```

end

```

Chapter 12

Test cases

```
theory Native-Word-Test imports  
  Uint64 Uint32 Uint16 Uint8 Uint Native-Cast-Uint  
  HOL-Library.Code-Test Word-Lib.Bit-Shifts-Infix-Syntax  
begin  
  
export-code  
  nat-of-uint8 uint8-of-nat  
  nat-of-uint16 uint16-of-nat  
  nat-of-uint32 uint32-of-nat  
  nat-of-uint64 uint64-of-nat  
  nat-of-uint uint-of-nat  
in SML
```

12.1 Tests for *uint32*

```
context  
  includes bit-operations-syntax  
begin  
  
abbreviation (input) sshiftr-uint32 (infixl >>> 55)  
  where  $\langle w \gg \rangle n \equiv \text{signed-drop-bit-uint32 } n \ w$ 
```

```
definition test-uint32 where  
  test-uint32  $\longleftrightarrow$   
  (([ 0x100000001,  $-1$ ,  $-4294967291$ , 0xFFFFFFFF, 0x12345678  
    , 0x5A AND 0x36  
    , 0x5A OR 0x36  
    , 0x5A XOR 0x36  
    , NOT 0x5A  
    ,  $5 + 6$ ,  $-5 + 6$ ,  $-6 + 5$ ,  $-5 + (-6)$ ,  $0xFFFFFFFF + 1$   
    ,  $5 - 3$ ,  $3 - 5$   
    ,  $5 * 3$ ,  $-5 * 3$ ,  $-5 * -4$ ,  $0x12345678 * 0x87654321$   
    ,  $5 \text{ div } 3$ ,  $-5 \text{ div } 3$ ,  $-5 \text{ div } -3$ ,  $5 \text{ div } -3$   
    ,  $5 \text{ mod } 3$ ,  $-5 \text{ mod } 3$ ,  $-5 \text{ mod } -3$ ,  $5 \text{ mod } -3$ 
```

```

, set-bit 5 4 True, set-bit (- 5) 2 True, set-bit 5 0 False, set-bit (- 5) 1 False
, set-bit 5 32 True, set-bit 5 32 False, set-bit (- 5) 32 True, set-bit (- 5) 32
False
, 1 << 2, -1 << 3, 1 << 32, 1 << 0
, 100 >> 3, -100 >> 3, 100 >> 32, -100 >> 32
, 100 >>> 3, -100 >>> 3, 100 >>> 32, -100 >>> 32] :: uint32 list)
=
[ 1, 4294967295, 5, 4294967295, 305419896
, 18
, 126
, 108
, 4294967205
, 11, 1, 4294967295, 4294967285, 0
, 2, 4294967294
, 15, 4294967281, 20, 1891143032
, 1, 1431655763, 0, 0
, 2, 2, 4294967291, 5
, 21, 4294967295, 4, 4294967289
, 5, 5, 4294967291, 4294967291
, 4, 4294967288, 0, 1
, 12, 536870899, 0, 0
, 12, 4294967283, 0, 4294967295]) ^
([ (0x5 :: uint32) = 0x5, (0x5 :: uint32) = 0x6
, (0x5 :: uint32) < 0x5, (0x5 :: uint32) < 0x6, (-5 :: uint32) < 6, (6 :: uint32)
< -5
, (0x5 :: uint32) ≤ 0x5, (0x5 :: uint32) ≤ 0x4, (-5 :: uint32) ≤ 6, (6 :: uint32)
≤ -5
, (0x7FFFFFFF :: uint32) < 0x80000000, (0xFFFFFFFF :: uint32) < 0,
(0x80000000 :: uint32) < 0x7FFFFFFF
, bit (0x7FFFFFFF :: uint32) 0, bit (0x7FFFFFFF :: uint32) 31, bit (0x80000000
:: uint32) 31, bit (0x80000000 :: uint32) 32
]
=
[ 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 0xA.A.A.A.A.A.A.A])
=
[0, 0x7FFFFFFF, 0x80000000, 0xA.A.A.A.A.A.A.A])

```

export-code test-uint32 checking SML Haskell? OCaml? Scala

notepad begin

have test-uint32 by eval

have test-uint32 by code-simp

```

have test-uint32 by normalization
end

```

```

definition test-uint32' :: uint32
where test-uint32' = 0 + 10 - 14 * 3 div 6 mod 3 << 3 >> 2
ML <val 0wx12 = @{code test-uint32'}>

```

```

lemma x AND y = x OR (y :: uint32)
quickcheck[random, expect=counterexample]
quickcheck[exhaustive, expect=counterexample]
oops

```

```

lemma (x :: uint32) AND x = x OR x
quickcheck[narrowing, expect=no-counterexample]
by transfer simp

```

```

lemma (f :: uint32 ⇒ unit) = g
quickcheck[narrowing, size=3, expect=no-counterexample]
by(simp add: fun-eq-iff)

```

```

end

```

12.2 Tests for *uint16*

```

context
  includes bit-operations-syntax
begin

```

```

abbreviation (input) sshiftr-uint16 (infixl >>> 55)
  where <w >>> n ≡ signed-drop-bit-uint16 n w

```

```

definition test-uint16 where
  test-uint16 <math>\longleftrightarrow</math>
  (([ 0x10001, -1, -65535, 0xFFFF, 0x1234
    , 0x5A AND 0x36
    , 0x5A OR 0x36
    , 0x5A XOR 0x36
    , NOT 0x5A
    , 5 + 6, -5 + 6, -6 + 5, -5 + -6, 0xFFFF + 1
    , 5 - 3, 3 - 5
    , 5 * 3, -5 * 3, -5 * -4, 0x1234 * 0x8765
    , 5 div 3, -5 div 3, -5 div -3, 5 div -3
    , 5 mod 3, -5 mod 3, -5 mod -3, 5 mod -3
    , set-bit 5 4 True, set-bit (- 5) 2 True, set-bit 5 0 False, set-bit (- 5) 1 False
    , set-bit 5 32 True, set-bit 5 32 False, set-bit (- 5) 32 True, set-bit (- 5) 32
    False
    , 1 << 2, -1 << 3, 1 << 16, 1 << 0
    , 100 >> 3, -100 >> 3, 100 >> 16, -100 >> 16
    , 100 >>> 3, -100 >>> 3, 100 >>> 16, -100 >>> 16] :: uint16 list)

```

```

=
[ 1, 65535, 1, 65535, 4660
, 18
, 126
, 108
, 65445
, 11, 1, 65535, 65525, 0
, 2, 65534
, 15, 65521, 20, 39556
, 1, 21843, 0, 0
, 2, 2, 65531, 5
, 21, 65535, 4, 65529
, 5, 5, 65531, 65531
, 4, 65528, 0, 1
, 12, 8179, 0, 0
, 12, 65523, 0, 65535]) ∧
((0x5 :: uint16) = 0x5, (0x5 :: uint16) = 0x6
, (0x5 :: uint16) < 0x5, (0x5 :: uint16) < 0x6, (-5 :: uint16) < 6, (6 :: uint16)
< -5
, (0x5 :: uint16) ≤ 0x5, (0x5 :: uint16) ≤ 0x4, (-5 :: uint16) ≤ 6, (6 :: uint16)
≤ -5
, (0x7FFF :: uint16) < 0x8000, (0xFFFF :: uint16) < 0, (0x8000 :: uint16) <
0x7FFF
, bit (0x7FFF :: uint16) 0, bit (0x7FFF :: uint16) 15, bit (0x8000 :: uint16)
15, bit (0x8000 :: uint16) 16
]
=
[ True, False
, False, True, False, True
, True, False, False, True
, True, False, False
, True, False, True, False
] ∧
([integer-of-uint16 0, integer-of-uint16 0x7FFF, integer-of-uint16 0x8000, inte-
ger-of-uint16 0xAAAA]
=
[0, 0x7FFF, 0x8000, 0xAAAA])

```

export-code *test-uint16 checking Haskell? Scala*

export-code *test-uint16 checking SML-word*

notepad begin

have *test-uint16* **by** *code-simp*

have *test-uint16* **by** *normalization*

end

lemma (*x :: uint16*) *AND x = x OR x*

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

by *transfer simp*

```

lemma (f :: uint16 ⇒ unit) = g
quickcheck[narrowing, size=3, expect=no-counterexample]
by(simp add: fun-eq-iff)

end

```

12.3 Tests for *uint8*

context

includes *bit-operations-syntax*

begin

abbreviation (*input*) *sshiftr-uint8* (**infixl** >>> 55)
where $\langle w \gg \rangle n \equiv \text{signed-drop-bit-uint8 } n \ w$

definition *test-uint8* **where**

```

test-uint8 ←→
([ [ 0x101, -1, -255, 0xFF, 0x12
  , 0x5A AND 0x36
  , 0x5A OR 0x36
  , 0x5A XOR 0x36
  , NOT 0x5A
  , 5 + 6, -5 + 6, -6 + 5, -5 + -6, 0xFF + 1
  , 5 - 3, 3 - 5
  , 5 * 3, -5 * 3, -5 * -4, 0x12 * 0x87
  , 5 div 3, -5 div 3, -5 div -3, 5 div -3
  , 5 mod 3, -5 mod 3, -5 mod -3, 5 mod -3
  , set-bit 5 4 True, set-bit (- 5) 2 True, set-bit 5 0 False, set-bit (- 5) 1 False
  , set-bit 5 32 True, set-bit 5 32 False, set-bit (- 5) 32 True, set-bit (- 5) 32
False
  , 1 << 2, -1 << 3, 1 << 8, 1 << 0
  , 100 >> 3, -100 >> 3, 100 >> 8, -100 >> 8
  , 100 >>> 3, -100 >>> 3, 100 >>> 8, -100 >>> 8] :: uint8 list)
=
[ 1, 255, 1, 255, 18
  , 18
  , 126
  , 108
  , 165
  , 11, 1, 255, 245, 0
  , 2, 254
  , 15, 241, 20, 126
  , 1, 83, 0, 0
  , 2, 2, 251, 5
  , 21, 255, 4, 249
  , 5, 5, 251, 251
  , 4, 248, 0, 1
  , 12, 19, 0, 0

```

```

    , 12, 243, 0, 255]) ^
  ([ (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**

have *test-uint8* **by** *eval*

have *test-uint8* **by** *code-simp*

have *test-uint8* **by** *normalization*

end

ML-val $\langle \text{val true} = @\{\text{code test-uint8}\} \rangle$

definition *test-uint8'* :: *uint8*

where *test-uint8'* = $0 + 10 - 14 * 3 \text{ div } 6 \text{ mod } 3 \ll 3 \gg 2$

ML $\langle \text{val } 0wx12 = @\{\text{code test-uint8}'\} \rangle$

lemma $x \text{ AND } y = x \text{ OR } (y :: \text{uint8})$

quickcheck[*random, expect=counterexample*]

quickcheck[*exhaustive, expect=counterexample*]

oops

lemma $(x :: \text{uint8}) \text{ AND } x = x \text{ OR } x$

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

by *transfer simp*

lemma $(f :: \text{uint8} \Rightarrow \text{unit}) = g$

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

by(*simp add: fun-eq-iff*)

12.4 Tests for *wint*

context

includes *bit-operations-syntax*

begin

abbreviation (*input*) *sshiftr-wint* (**infixl** >>> 55)

where $\langle w \gg \rangle n \equiv \text{signed-drop-bit-wint } n \ w$

definition *test-wint* \equiv *let*

test-list1 = (*let*

HS = *wint-of-int* ($2 \wedge (\text{dflt-size} - 1)$)

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
 , set-bit 5 4 True, set-bit (- 5) 2 True, set-bit 5 0 False, set-bit (- 5) 1 False
 , set-bit 5 *dflt-size* True, set-bit 5 *dflt-size* False, set-bit (- 5) *dflt-size* True,
 set-bit (- 5) *dflt-size* False

, 1 << 2, -1 << 3, push-bit *dflt-size* 1, 1 << 0
 , 31 >> 3, -1 >> 3, 31 >> *dflt-size*, -1 >> *dflt-size*
 , 15 >>> 2, -1 >>> 3, 15 >>> *dflt-size*, -1 >>> *dflt-size*]

else [] :: *wint 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)
 , set-bit 5 4 True, -1, set-bit 5 0 False, -7
 , 5, 5, -5, -5
 , 4, -8, 0, 1

```

    , 3, (S >> 3) - 1, 0, 0
    , 3, (S >> 1) + (S >> 1) - 1, 0, -1]
  else [] :: int list));

test-list-c1 = (let
  HS = uint-of-int ((2^(dflt-size - 1)))
  in
  [ (0x5 :: uint) = 0x5, (0x5 :: uint) = 0x6
  , (0x5 :: uint) < 0x5, (0x5 :: uint) < 0x6, (-5 :: uint) < 6, (6 :: uint) < -5
  , (0x5 :: uint) ≤ 0x5, (0x5 :: uint) ≤ 0x4, (-5 :: uint) ≤ 6, (6 :: uint) ≤ -5
  , (HS - 1) < HS, (HS + HS - 1) < 0, HS < HS - 1
  , bit (HS - 1) 0, bit (HS - 1 :: uint) (dflt-size - 1), bit (HS :: uint) (dflt-size
  - 1), bit (HS :: uint) dflt-size
  ]);

test-list-c2 =
  [ True, False
  , False, dflt-size ≥ 2, dflt-size = 3, dflt-size ≠ 3
  , True, False, dflt-size = 3, dflt-size ≠ 3
  , True, False, False
  , dflt-size ≠ 1, False, True, False
  ]
in
  test-list1 = map uint-of-int test-list2
  ∧ test-list-c1 = test-list-c2

export-code test-uint checking SML Haskell? OCaml? Scala

lemma test-uint
quickcheck[exhaustive, expect=no-counterexample]
oops — FIXME: prove correctness of test by reflective means (not yet supported)

lemma x AND y = x OR (y :: uint)
quickcheck[random, expect=counterexample]
quickcheck[exhaustive, expect=counterexample]
oops

lemma (x :: uint) AND x = x OR x
quickcheck[narrowing, expect=no-counterexample]
by transfer simp

lemma (f :: uint ⇒ unit) = g
quickcheck[narrowing, size=3, expect=no-counterexample]
by(simp add: fun-eq-iff)

```

12.5 Tests for *uint64*

```

context
  includes bit-operations-syntax

```

begin

abbreviation (*input*) *sshiftr-uint64* (**infixl** >>> 55)

where $\langle w \gg \rangle n \equiv \text{signed-drop-bit-uint64 } n \ w$

definition *test-uint64* **where**

test-uint64 \longleftrightarrow
 ([0x10000000000000001, -1, -9223372036854775808, 0xFFFFFFFFFFFFFFFF, 0x1234567890ABCDEF
 , 0x5A AND 0x36
 , 0x5A OR 0x36
 , 0x5A XOR 0x36
 , NOT 0x5A
 , 5 + 6, -5 + 6, -6 + 5, -5 + (- 6), 0xFFFFFFFFFFFFFFFF + 1
 , 5 - 3, 3 - 5
 , 5 * 3, -5 * 3, -5 * -4, 0x1234567890ABCDEF * 0xFEDCBA0987654321
 , 5 div 3, -5 div 3, -5 div -3, 5 div -3
 , 5 mod 3, -5 mod 3, -5 mod -3, 5 mod -3
 , set-bit 5 4 True, set-bit (- 5) 2 True, set-bit 5 0 False, set-bit (- 5) 1 False
 , set-bit 5 64 True, set-bit 5 64 False, set-bit (- 5) 64 True, set-bit (- 5) 64
 False
 , 1 << 2, -1 << 3, 1 << 64, 1 << 0
 , 100 >> 3, -100 >> 3, 100 >> 64, -100 >> 64
 , 100 >>> 3, -100 >>> 3, 100 >>> 64, -100 >>> 64] :: uint64 list)
 =
 [1, 18446744073709551615, 9223372036854775808, 18446744073709551615,
 1311768467294899695
 , 18
 , 126
 , 108
 , 18446744073709551525
 , 11, 1, 18446744073709551615, 18446744073709551605, 0
 , 2, 18446744073709551614
 , 15, 18446744073709551601, 20, 14000077364136384719
 , 1, 6148914691236517203, 0, 0
 , 2, 2, 18446744073709551611, 5
 , 21, 18446744073709551615, 4, 18446744073709551609
 , 5, 5, 18446744073709551611, 18446744073709551611
 , 4, 18446744073709551608, 0, 1
 , 12, 2305843009213693939, 0, 0
 , 12, 18446744073709551603, 0, 18446744073709551615] \wedge
 ([(0x5 :: uint64) = 0x5, (0x5 :: uint64) = 0x6
 , (0x5 :: uint64) < 0x5, (0x5 :: uint64) < 0x6, (-5 :: uint64) < 6, (6 :: uint64)
 < -5
 , (0x5 :: uint64) \leq 0x5, (0x5 :: uint64) \leq 0x4, (-5 :: uint64) \leq 6, (6 :: uint64)
 \leq -5
 , (0x7FFFFFFFFFFFFFFFFF :: uint64) < 0x8000000000000000, (0xFFFFFFFFFFFFFFFF
 :: uint64) < 0, (0x8000000000000000 :: uint64) < 0x7FFFFFFFFFFFFFFFFF
 , bit (0x7FFFFFFFFFFFFFFFFF :: uint64) 0, bit (0x7FFFFFFFFFFFFFFFFF ::

```

uint64) 63, bit (0x8000000000000000 :: uint64) 63, bit (0x8000000000000000 ::
uint64) 64
]
=
[ True, False
, False, True, False, True
, True, False, False, True
, True, False, False
, True, False, True, False
] ^
([integer-of-uint64 0, integer-of-uint64 0x7FFFFFFFFFFFFFFF, integer-of-uint64
0x8000000000000000, integer-of-uint64 0xAAAAAAAAAAAAAAAA]
=
[0, 0x7FFFFFFFFFFFFFFF, 0x8000000000000000, 0xAAAAAAAAAAAAAAAA])

```

```

value [nbe] [0x10000000000000001, -1, -9223372036854775808, 0xFFFFFFFFFFFFFFFF,
0x1234567890ABCDEF
, 0x5A AND 0x36
, 0x5A OR 0x36
, 0x5A XOR 0x36
, NOT 0x5A
, 5 + 6, -5 + 6, -6 + 5, -5 + (- 6), 0xFFFFFFFFFFFFFFFF + 1
, 5 - 3, 3 - 5
, 5 * 3, -5 * 3, -5 * -4, 0x1234567890ABCDEF * 0xFEDCBA0987654321
, 5 div 3, -5 div 3, -5 div -3, 5 div -3
, 5 mod 3, -5 mod 3, -5 mod -3, 5 mod -3
, set-bit 5 4 True, set-bit (- 5) 2 True, set-bit 5 0 False, set-bit (- 5) 1 False
, set-bit 5 64 True, set-bit 5 64 False, set-bit (- 5) 64 True, set-bit (- 5) 64
False
, 1 << 2, -1 << 3, 1 << 64, 1 << 0
, 100 >> 3, -100 >> 3, 100 >> 64, -100 >> 64
, 100 >>> 3, -100 >>> 3, 100 >>> 64, -100 >>> 64] :: uint64 list

```

```

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

```

```

notepad begin

```

```

have test-uint64 by eval

```

```

have test-uint64 by code-simp

```

```

have test-uint64 by normalization

```

```

end

```

```

ML-val <val true = @{code test-uint64}>

```

```

definition test-uint64' :: uint64

```

```

where test-uint64' = 0 + 10 - 14 * 3 div 6 mod 3 << 3 >> 2

```

```

end

```

12.6 Tests for casts

definition *test-casts* :: *bool*

where *test-casts* \longleftrightarrow

map uint8-of-uint32 [10, 0, 0xFE, 0xFFFFFFFF] = [10, 0, 0xFE, 0xFF] \wedge
map uint8-of-uint64 [10, 0, 0xFE, 0xFFFFFFFFFFFFFFFF] = [10, 0, 0xFE, 0xFF] \wedge
map uint32-of-uint8 [10, 0, 0xFF] = [10, 0, 0xFF] \wedge
map uint64-of-uint8 [10, 0, 0xFF] = [10, 0, 0xFF]

definition *test-casts'* :: *bool*

where *test-casts'* \longleftrightarrow

map uint8-of-uint16 [10, 0, 0xFE, 0xFFFF] = [10, 0, 0xFE, 0xFF] \wedge
map uint16-of-uint8 [10, 0, 0xFF] = [10, 0, 0xFF] \wedge
map uint16-of-uint32 [10, 0, 0xFFFE, 0xFFFFFFFF] = [10, 0, 0xFFFE, 0xFFFF] \wedge
map uint16-of-uint64 [10, 0, 0xFFFE, 0xFFFFFFFFFFFFFFFF] = [10, 0, 0xFFFE, 0xFFFF] \wedge
map uint32-of-uint16 [10, 0, 0xFFFF] = [10, 0, 0xFFFF] \wedge
map uint64-of-uint16 [10, 0, 0xFFFF] = [10, 0, 0xFFFF]

definition *test-casts''* :: *bool*

where *test-casts''* \longleftrightarrow

map uint32-of-uint64 [10, 0, 0xFFFFFFFFFE, 0xFFFFFFFFFFFFFFFF] = [10, 0, 0xFFFFFFFFFE, 0xFFFFFFFF] \wedge
map uint64-of-uint32 [10, 0, 0xFFFFFFFF] = [10, 0, 0xFFFFFFFF]

export-code *test-casts test-casts'' checking SML Haskell? Scala*

export-code *test-casts'' checking OCaml?*

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

notepad begin

have *test-casts* **by** *eval*
have *test-casts* **by** *normalization*
have *test-casts* **by** *code-simp*
have *test-casts'* **by** *normalization*
have *test-casts'* **by** *code-simp*
have *test-casts''* **by** *eval*
have *test-casts''* **by** *normalization*
have *test-casts''* **by** *code-simp*

end

ML \langle

val true = @{code *test-casts*}
val true = @{code *test-casts''*}

\rangle

definition *test-casts-uint* :: *bool* **where**

test-casts-uint \longleftrightarrow

map uint-of-uint32 ([0, 10] @ (if *dflt-size* < 32 then [*push-bit* (*dflt-size* - 1) 1,

```

0xFFFFFFFF] else [0xFFFFFFFF])) =
  [0, 10] @ (if dflt-size < 32 then [push-bit (dflt-size - 1) 1, (push-bit dflt-size 1)
- 1] else [0xFFFFFFFF]) ^
  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]

```

end

end

end

Chapter 13

Implementation of bit operations on int by target language operations

```
theory Code-Target-Bits-Int
  imports
    Bits-Integer
    HOL-Library.Code-Target-Int
begin

context
  includes bit-operations-syntax
begin

declare [[code drop:
  (AND) :: int ⇒ -   (OR) :: int ⇒ -   (XOR) :: int ⇒ -   NOT :: int ⇒ -
  lsb :: int ⇒ -   set-bit :: int ⇒ -   bit :: int ⇒ -
  push-bit :: - ⇒ int ⇒ -   drop-bit :: - ⇒ int ⇒ -
  int-of-integer-symbolic
]]

lemma [code-unfold]:
  ⟨of-bool (odd i) = i AND 1⟩ for i :: int
  by (simp add: and-one-eq mod2-eq-if)

lemma [code-unfold]:
  ⟨bit x n ⟷ x AND (push-bit n 1) ≠ 0⟩ for x :: int
  by (fact bit-iff-and-push-bit-not-eq-0)

context
  includes integer.lifting
begin
```

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

bit (int-of-integer x) n = bit x n

by *transfer simp*

lemma *and-int-code* [code]:

int-of-integer i AND int-of-integer j = int-of-integer (i AND j)

by *transfer simp*

lemma *or-int-code* [code]:

int-of-integer i OR int-of-integer j = int-of-integer (i OR j)

by *transfer simp*

lemma *xor-int-code* [code]:

int-of-integer i XOR int-of-integer j = int-of-integer (i XOR j)

by *transfer simp*

lemma *not-int-code* [code]:

NOT (int-of-integer i) = int-of-integer (NOT i)

by *transfer simp*

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

⟨push-bit n (int-of-integer x) = int-of-integer (push-bit n x)⟩

by *transfer simp*

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

⟨drop-bit n (int-of-integer x) = int-of-integer (drop-bit n x)⟩

by *transfer simp*

lemma *take-bit-int-code* [code]:

⟨take-bit n (int-of-integer x) = int-of-integer (take-bit n x)⟩

by *transfer simp*

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

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

by *transfer simp*

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

set-bit (int-of-integer x) n b = int-of-integer (set-bit x n b)

by *transfer simp*

lemma *int-of-integer-symbolic-code* [code]:

int-of-integer-symbolic = int-of-integer

by (*simp add: int-of-integer-symbolic-def*)

context

begin

qualified definition *even* :: *⟨int ⇒ bool⟩*

where [code-abbrev]: *⟨even = Parity.even⟩*

end

lemma *[code]*:

$\langle \text{Code-Target-Bits-Int.even } i \longleftrightarrow i \text{ AND } 1 = 0 \rangle$

by (*simp add: Code-Target-Bits-Int.even-def even-iff-mod-2-eq-zero and-one-eq*)

lemma *bin-rest-code*:

$\text{int-of-integer } i \text{ div } 2 = \text{int-of-integer } (\text{bin-rest-integer } i)$

by *transfer simp*

end

end

end

theory *Native-Word-Test-Emu* **imports**

Native-Word-Test

Code-Target-Bits-Int

begin

13.1 Test cases for emulation of native words

13.1.1 Tests for *uint16*

Test that *uint16* is emulated for PolyML and OCaml via *16 word* if *Native-Word.Code-Target-Bits-Int* is imported.

definition *test-uint16-emulation* :: *bool* **where**

$\text{test-uint16-emulation} \longleftrightarrow (0xFFFF - 0x1000 = (0xEFFF :: \text{uint16}))$

export-code *test-uint16-emulation* **checking** *SML OCaml?*

— test the other target languages as well *Haskell? Scala*

notepad **begin**

have *test-uint16* **by** *eval*

have *test-uint16-emulation* **by** *eval*

have *test-uint16-emulation* **by** *normalization*

have *test-uint16-emulation* **by** *code-simp*

end

ML-val \langle

val true = $\text{@}\{\text{code } \text{test-uint16}\};$

val true = $\text{@}\{\text{code } \text{test-uint16-emulation}\};$

\rangle

lemma $x \text{ AND } y = x \text{ OR } (y :: \text{uint16})$

```

quickcheck[random, expect=counterexample]
quickcheck[exhaustive, expect=counterexample]
oops

```

13.1.2 Tests for *uint8*

Test that *uint8* is emulated for OCaml via *8 word* if *Native-Word.Code-Target-Bits-Int* is imported.

```

definition test-uint8-emulation :: bool where
  test-uint8-emulation  $\longleftrightarrow$  (0xFFF - 0x10 = (0xEF :: uint8))

```

```

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

```

```

end

```

```

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

```

13.2 Test with PolyML

```

test-code
  test-uint64 test-uint64' = 0x12
  test-uint32 test-uint32' = 0x12
  test-uint8 test-uint8' = 0x12
  test-uint
  test-casts test-casts''
  test-casts-uint test-casts-uint''
in PolyML

```

```

end

```

```

theory Native-Word-Test-PolyML2 imports
  Native-Word-Test-Emu
begin

```

```

test-code
  test-uint16 test-uint16-emulation
  test-casts'
  test-casts-uint'
in PolyML

```

```

end

```

```

theory Native-Word-Test-PolyML64 imports
  Native-Word-Test
begin

test-code test-uint64' = 0x12
in PolyML

ML ⟨
  if ML-System.platform-is-64 then
    ML ⟨assert (code ⟨test-uint64' = 0wx12)⟩
  else ()
  ⟩

end

```

```

theory Native-Word-Test-Scala imports
  Native-Word-Test
begin

```

13.3 Test with Scala

In Scala, *uint* and *uint32* are both implemented as type `Int`. When they are used in the same generated program, we have to suppress the type class instances for one of them.

code-printing class-instance *uint32* :: *equal* \rightarrow (*Scala*) –

```

test-code
  test-uint64 test-uint64' = 0x12
  test-uint32 test-uint32' = 0x12
  test-uint16
  test-uint8 test-uint8' = 0x12
  test-uint
  test-casts test-casts' test-casts''
  test-casts-uint test-casts-uint' test-casts-uint''
in Scala

end

```


Chapter 14

User guide for native words

This tutorial explains how to best use the types for native words like *wint32* in your formalisation. You can base your formalisation

1. either directly on these types,
2. or on the generic *'a word* and only introduce native words a posteriori via code generator refinement.

The first option causes the least overhead if you have to prove only little about the words you use and start a fresh formalisation. Just use the native type *wint32* instead of *32 word* and similarly for *wint64*, *wint16*, and *wint8*. As native word types are meant only for code generation, the lemmas about *'a word* have not been duplicated, but you can transfer theorems between native word types and *'a word* using the transfer package.

Note, however, that this option restricts your work a bit: your own functions cannot be “polymorphic” in the word length, but you have to define a separate function for every word length you need.

The second option is recommended if you already have a formalisation based on *'a word* or if your proofs involve words and their properties. It separates code generation from modelling and proving, i.e., you can work with words as usual. Consequently, you have to manually setup the code generator to use the native types wherever you want. The following describes how to achieve this with moderate effort.

Note, however, that some target languages of the code generator (especially OCaml) do not support all the native word types provided. Therefore, you should only import those types that you need – the theory file for each type mentions at the top the restrictions for code generation. For example, PolyML does not provide the *Word16* structure, and OCaml provides neither *Word8* nor *Word16*. You can still use these theories provided that you also import the theory *Native-Word.Code-Target-Bits-Int* (which implements *int*

by target-language integers), but these words will be implemented via Isabelle's *Word* library, i.e., you do not gain anything in terms of efficiency.

There is a separate code target *SML-word* for SML. If you use one of the native words that PolyML does not support (such as *uint16* and *uint64* in 32-bit mode), but would like to map its operations to the Standard Basis Library functions, make sure to use the target *SML-word* instead of *SML*; if you only use native word sizes that PolyML supports, you can stick with *SML*. This ensures that code generation within Isabelle as used by *Quickcheck*, *value* and `@{code}` in ML blocks continues to work.

14.1 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 + $n * n$) ($n - 1$))

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* .

lift-definition *sum-squares-uint* :: *uint16* \Rightarrow *uint16* is *sum-squares* .

lift-definition *sum-squares-int-uint* :: int \Rightarrow *uint16* is *sum-squares-int* .

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
```

14.2 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.

14.2.1 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  $\Rightarrow$  expr where word i  $\equiv$  Lit (Word i)
```

```
abbreviation (input) bool :: bool  $\Rightarrow$  expr where bool i  $\equiv$  Lit (Bool i)
```

— Denotational semantics of expressions, *None* denotes a type error

```
fun eval :: expr  $\Rightarrow$  val option where
  eval (Lit v) = Some v
| eval (LT e1 e2) =
  (case (eval e1, eval e2)
   of (Some (Word i1), Some (Word i2))  $\Rightarrow$  Some (Bool (i1 < i2)))
  | -  $\Rightarrow$  None
| eval (IF e1 e2 e3) =
```

(*case eval* e_1 of *Some* (*Bool* b) \Rightarrow if b then *eval* e_2 else *eval* e_3
 | - \Rightarrow *None*)

— Small-step semantics of expressions, it gets stuck upon type errors.

inductive step :: *expr* \Rightarrow *expr* \Rightarrow *bool* (- \rightarrow - [50, 50] 60) **where**
 | $e \rightarrow e' \Rightarrow$ *LT* e $e_2 \rightarrow$ *LT* e' e_2
 | $e \rightarrow e' \Rightarrow$ *LT* (*word* i) $e \rightarrow$ *LT* (*word* i) e'
 | *LT* (*word* i_1) (*word* i_2) \rightarrow *bool* ($i_1 < i_2$)
 | $e \rightarrow e' \Rightarrow$ *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* .

14.2.2 Change the datatype to use machine words

Now, we want to use *uint32* instead of *32 word*. The goal is to make the code generator use the new type without duplicating any of the types (*val*, *expr*) or the functions (*eval*, *reduce*) on such types.

The constructor *Word* has *32 word* in its type, so we have to lift it to *Word'*, and the same holds for the case combinator *case-val*, which *case-val'* replaces.¹ Next, we set up the code generator accordingly: *Bool* and *Word'* are the new constructors for *val*, and *case-val'* is the new case combinator with an appropriate case certificate.² We delete the code equations for the old constructor *Word* and case combinator *case-val* such that the code generator reports missing adaptations.

lift-definition *Word'* :: *uint32* \Rightarrow *val* is *Word* .

code-datatype *Bool* *Word'*

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

²Case certificates tell the code generator to replace the HOL case combinator for a datatype with the case combinator of the target language. Without a case certificate, the code generator generates a function that re-implements the case combinator; in a strict languages like ML or Scala, this means that the code evaluates all possible cases before it decides which one is taken.

Case certificates are described in Haftmann's PhD thesis [1, Def. 27]. For a datatype *dt* with constructors C_1 to C_n where each constructor C_i takes k_i parameters, the certificate for the case combinator *case-dt* looks as follows:

lemma

assumes *CASE* \equiv *dt-case* c_1 c_2 ... c_n

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

&&& (*CASE* (C_2 a_{21} a_{22} ... a_{2k_2}) \equiv c_2 a_{21} a_{22} ... a_{2k_2})

&&& ...

&&& (*CASE* (C_n a_{n1} a_{n2} ... a_{nk_n}) \equiv c_n a_{n1} a_{n2} ... a_{nk_n})

lift-definition *case-val'* :: (bool \Rightarrow 'a) \Rightarrow (uint32 \Rightarrow 'a) \Rightarrow val \Rightarrow 'a **is** *case-val* .

lemmas [*code*, *simp*] = val.case [Transfer.transferred]

lemma *case-val'-cert*:

fixes bool word' b w

assumes CASE \equiv *case-val'* bool word'

shows (CASE (Bool b) \equiv bool b) &&& (CASE (Word' w) \equiv word' w)

by (*simp-all add: assms*)

setup <Code.declare-case-global @{thm *case-val'-cert*}>

declare [[*code drop: case-val Word*]]

14.2.3 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*

14.3 Troubleshooting

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

14.3.1 *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** $\lambda x. x + x$.

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*
by *transfer simp*

14.3.2 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 §14.3.1. For Haskell, you have to enable the extension `TypeSynonymInstances` with `-XTypeSynonymInstances` if you are using polymorphic bit operations on the native word types.

14.3.3 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.