Isabelle Theories for Machine Words

Jeremy Dawson

Logic and Computation Program, NICTA [1]

Automated Reasoning Group,
Australian National University, Canberra, ACT 0200, Australia
http://users.rsise.anu.edu.au/~jeremy/

September 4, 2007

[1] National ICT Australia is funded by the Australian Government's Dept of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Centre of Excellence program.

Outline

1. Introduction
   - The Isabelle theorem prover
   - Comparing Related Work
2. The word-n theories
   - Numerical n-bit quantities: the `bin` and `obin` types
     - Using datatype-like properties of `bins`
   - The type of fixed-length words of given length
   - Sets isomorphic to the set of words
   - Simplifications for arithmetic expressions
   - Miscellaneous techniques

Outline

1. Introduction
   - The Isabelle theorem prover
   - Comparing Related Work
2. The word-n theories
   - Numerical n-bit quantities: the `bin` and `obin` types
     - Using datatype-like properties of `bin`s
   - The type of fixed-length words of given length
   - Sets isomorphic to the set of words
   - Simplifications for arithmetic expressions
   - Miscellaneous techniques

Introduction

NICTA's L4.verified project: to provide a mathematical, machine-checked proof of the correctness of the L4 microkernel

In formally verifying machine hardware, we need to be able to systematically deal with the properties of machine words. These differ from ordinary numbers in that, for example,

- addition and multiplication can overflow, with overflow bits being lost,
- and there are bit-wise operations which are simply defined in a natural way.

The Isabelle theorem prover

- Logical framework: logic ("meta-logic") is intuitionistic polymorphically-typed higher-order logic
- Choice of "object logic": we use HOL, "Higher-Order Logic":
    - uses type system of meta-logic
    - classical
    - Axiom of Choice
- This HOL object logic inspired by HOL theorem prover
- Both Isabelle and HOL are LCF-based, written in Standard ML
- User interaction via Standard ML or Isar

Related Work in the HOL prover

Wai Wong

- words are lists of bits.
- The type is all words of any length;
- Some theorems conditional on word length
- Bit-wise operations, but no arithmetic operations.

Related Work in the HOL prover

Wai Wong

- words are lists of bits.
- The type is all words of any length;
- Some theorems conditional on word length
- Bit-wise operations, but no arithmetic operations.

Fox

- machine word type is isomorphic to the naturals,
- `W32` n is the word with unsigned value $n \bmod 2^{32}$.
- equality of machine words is *not* equality of their representations.

Related Work in the HOL prover

Wai Wong

- words are lists of bits.
- The type is all words of any length;
- Some theorems conditional on word length
- Bit-wise operations, but no arithmetic operations.

Fox

- machine word type is isomorphic to the naturals,
- `W32` n is the word with unsigned value n mod 2^{32}.
- equality of machine words is *not* equality of their representations.

Harrison

- encodes vectors of dimension n of (reals, bits, etc)
- a type cannot be parameterised over the value n.
- uses type $N \rightarrow A$, where N is a *type* with exactly n values.

Other Related Work

PVS

- in PVS, a type *can* be parameterised over a value n
- a bit-vector is a function from $\{0, \ldots, N-1\}$ to the booleans
- PVS bit-vector library provides interpretations of a bit-vector as unsigned or signed integers
- may be better when concatenating or splitting words (involving words of length n, m, $n+m$)

Our Formalisation

- each type of words in our formalization is of a given length.
- word types related to integers mod 2^n and to lists of booleans
- many results re arithmetic and logical (bit-wise) operations.
- recent collaboration with Galois Connections
  (theirs more general: integers modulo m, for ours $m = 2^n$).
- Lots of operations on words which are not discussed here
- Isabelle code files are available

Outline

1. Introduction
   - The Isabelle theorem prover
   - Comparing Related Work

2. The word-n theories
   - Numerical n-bit quantities: the `bin` and `obin` types
     - Using datatype-like properties of `bins`
   - The type of fixed-length words of given length
   - Sets isomorphic to the set of words
   - Simplifications for arithmetic expressions
   - Miscellaneous techniques

the `bin` type

Isabelle's `bin` type explicitly represents bit strings, important as

- used for encoding numbers literally, an integer entered is converted to a `bin`, thus `read "3"` gives `number_of (Pls BIT B1 BIT B1 :: bin)`
- much built-in numeric simplification for numbers expressed as `bins`, for example for negation, addition and multiplication, using usual rules for twos-complement integers.

the old and new `bin` types

Isabelle had changed: formerly `bin` was a datatype: constructors

- `Pls` (a sequence of 0, extending infinitely leftwards)
- `Min` (a sequence of 1, extending infinitely leftwards) (for the integer -1)
- `BIT` (where `(w::bin) BIT (b::bool)` is `w` with `b` appended on the right)

Now call these `oPls, oMin, OBIT`, for the datatype `obin`.

After the change (in Isabelle 2005) `bin` is an abstract type, isomorphic to the set of all integers
w BIT $b = 2w + b$ `Pls` $= 0$ `Min` $=$ -1

Natural definitions using the obin datatype

Using obin datatype allows natural definition of functions by their action on bits

```
primrec
   obin_not_Pls :  "obin_not oPls = oMin"
   obin_not_Min :  "obin_not oMin = oPls"
   obin_not_OBIT :
       "obin_not (w OBIT x) = (obin_not w OBIT Not x)"
```

Defining arithmetic operations: close to twos-complement arithmetic as in the hardware

Easy to be sure that it is accurate: this is important for formal verification!!

Normalising `obins`

We normalise an `obin` by changing `oPls OBIT False` to `oPls`, as they represent the same sequence of bits
and likewise `oMin OBIT True` to `oMin`.

Set of normalised `obins` isomorphic to the set of integers,
via the usual twos-complement representation (PROVE IT!)

This issue added to the complexity of using `obins`

More problems of using the obin type

need to deal with words entered literally: `6 :: 'a word` is read as `number_of (Pls BIT B1 BIT B1 BIT B0)`

need simplifications for bit-wise (eg) conjunction of such `bins`

As `bin` is not a datatype, we first defined `bin_and` from `obin_and`

```
bin_and_def :  "bin_and v w ==
     onum_of (obin_and (int_to_obin v, int_to_obin w))"
```

Lots of simplification theorems about `obins` had to be transferred to `bins` — complex programming required

Using datatype-like properties of `bins`

Want to define functions in terms of the bit-representation of a `bin`

What properties of `bin` type resemble properties of a datatype?

The properties of a datatype are:

1. Different constructors give distinct values
2. Each constructor is injective (in each of its arguments)
3. All values of the type are obtained using the constructors

consider `bin` type with "pseudo-constructors" `Pls`, `Min` and `BIT`

In terms of these "pseudo-constructors" 2 and 3 above hold: in fact 3 holds using `BIT` alone

Defining functions on bins

Those properties give these theorems; `bin_exhaust` enables us to express any `bin` appearing in a proof as `w BIT b`

```
BIT_eq = "u BIT b = v BIT c ==> u = v & b = c"
bin_exhaust = "(!!x b.  bin = x BIT b ==> Q) ==> Q"
```

```
bin_rl_def :  "bin_rl w == SOME (r, l). w = r BIT l"
```

Since there is a unique choice of `r` and `l` to satisfy `w = r BIT l`, this means that `bin_rl (r BIT l) = (r, l)`

Induction principle for `bins`:

```
bin_induct = "[| P Pls; P Min;
  !!bin bit.  P bin ==> P (bin BIT bit) |] ==> P bin"
```

Imitating primitive recursion for bins

To define a function `f` by primitive recursion,
if `bin` were a datatype with its three constructors, require

- values `vp` and `vn` for `f Pls` and `f Min`,
- a function `fr`, where `f (w BIT b)` is given by `fr w b (f w)`

So, using Isabelle's `recdef` (for recursive functions), we defined

$$\mathtt{bin\_rec} : \alpha \to \alpha \to (\mathtt{int} \to \mathtt{bit} \to \alpha \to \alpha) \to \mathtt{int} \to \alpha$$

which, given `vp` `vn` and `fr`, returns a function `f` satisfying

```
f Pls = vp
f Min = vn
```

and, except where `w BIT b` equals `Pls` or `Min`,

```
f (w BIT b) = fr w b (f w)
```

Usually we can prove that this last equation holds for *all* `w` and `b`

Examples of definitions on bins

```
bin_not_def :  "bin_not == bin_rec Min Pls
  (%w b s. s BIT bit_not b)"
```

After making these definitions, the simplification rules in the desired form (such as those shown below) need to be proved.

```
bin_not_simps = [...  ,
   "bin_not (w BIT b) = bin_not w BIT bit_not b" ]
```

Proving these was fairly straightforward

Examples of definitions on bins

```
bin_not_def :  "bin_not == bin_rec Min Pls
  (%w b s. s BIT bit_not b)"
```

After making these definitions, the simplification rules in the desired form (such as those shown below) need to be proved.

```
bin_not_simps = [...   ,
   "bin_not (w BIT b) = bin_not w BIT bit_not b" ]
```

Proving these was fairly straightforward

```
bin_and_def :  "bin_and == bin_rec (%x. Pls) (%y. y)
(%w b s y. s (bin_rest y) BIT bit_and b (bin_last y))"
```

```
bin_and_Bits = "bin_and (x BIT b) (y BIT c) =
   bin_and x y BIT bit_and b c"
```

A type for n-bit quantities

Need to set up a type in which the length of words is implicit.

dependent types not allowed: lists of length n cannot be a type

Our solution: the type of words of length n is α `word`

where the word length can be deduced from the type α.

We use `len_of TYPE(`α`)` for the word length. `TYPE(`α`)` is a polymorphic value (a "canonical" value for each type)

```
len_of :: "'a :: len0 itself => nat"
word_size : "size (w :: 'a word) = len_of TYPE('a)"
```

user must define the value of `len_of TYPE(`α`)` for each specific α.

Constructing n-bit quantities; the type definition:

"truncation" functions `bintrunc` (unsigned),
and `sbintrunc` (signed) to create n-bit quantities.

- cut down a longer argument by deleting high-order bits.
- extend a shorter argument it to the left with zeroes (unsigned) or its most significant bit (signed)

Isabelle `typedef` defines a new type isomorphic to a given set.

```
typedef 'a word = "uword_len (len_of TYPE('a))"
"uword_len len == range (bintrunc len)"
```

Isomorphisms of set of words

type of words of length n defined isormorphic to `range (bintrunc n)`, but also isomorphic to the set of

- integers in the range $0 \ldots 2^n - 1$
- integers in the range $-2^{n-1} \ldots 2^{n-1} - 1$
- naturals up to $2^n - 1$
- lists of booleans of length n
- functions $f : nat \rightarrow bool$ such that for $i \geq n$, $f\ i = \mathtt{False}$

Pseudo type definition theorems

defining new type α from $S : \rho$ *set* gives

`Abs` $: \rho \rightarrow \alpha$ and `Rep` $: \alpha \rightarrow \rho$:

mutually inverse bijections between S and the values of type α

nothing known about values of `Abs` outside S

Theorem (axiom) `type_definition_`α created for the new type α
`type_definition Rep Abs` S

We can use the predicate `type_definition` to express the other isomophisms of the set of n-bit words mentioned above

We used SML functors to prove a collection of useful consequences of each such isomophism (can also use locales)

Extended type definition theorems

type definition theorems do not say anything about the action of `Abs` outside the set S

But our `Abs` functions behave "sensibly" outside S

Thus `word_of_int` (ie, `Abs`) which turns an integer in $0 \ldots 2^n - 1$ into a word, takes i and i' to the same word iff $i \equiv i' \pmod{2^n}$

Call `Rep` $\circ$ `Abs` *normalise*, `norm`. (eg, `norm` $i = i \pmod{2^n}$)

Say x is normal if $x =$ `norm` y for some y, iff $x =$ `norm` x

In many cases, have extended "extended type definition theorems", of the form `td_ext Rep Abs A norm`

Generated numerous results from each of these eg `norm` $\circ$ `Rep` $=$ `Rep`, and `Abs` $\circ$ `norm` $=$ `Abs`

Simplifications for arithmetic expressions

Certain arithmetic equalities hold for words, eg associativity and commutativity of addition and multiplication and distributivity of multiplication over addition

Single function `int2lenw` in Standard ML to generate these from corresponding results for integers

showed word type in many of Isabelle's arithmetical *type classes*

Therefore many automatic simplifications for these type classes are available for the word type

Thus `a + b + c = (b + d :: 'a :: len0 word)` is simplified to `a + c = d` (uses Isabelle's *simplification procedures*)

Simplifications of literals

Literal numbers syntax-translated, eg
5 becomes `number_of (Pls BIT B1 BIT B0 BIT B1)`

For words, define function `number_of` by
`"number_of (w::bin) :: 'a::len0 word == word_of_int w"`

Isabelle simplifies arithmetic expressions involving literal words by binary arithmetic (requires word type in class `number_ring`)

Thus `(6 + 5 :: 'a :: len word)` gets simplified to 11 automatically, regardless of the word length

Further simplification from `(11 :: word2)` to 3 and from `iszero (4 :: word2)` to `True` depend on the specific word length

Simplifications for bit-wise (logical) operations depend on simplifications for `bin_and`, `bin_not`, etc (discussed earlier)

One ML function translates many logical identities on `bin`s to words

Special-purpose simplification tactics

result (for words) `"(x < x - z) = (x < z)"`:
each inequality holds iff calculating $x - z$ causes underflow

tactic `uint_pm_tac` useful for such goals:

- unfolds definitions, gets goal using `uint x`, `uint z` (integers) and case analysis (if $z \leq x$ then ... else ...)
- for every `uint` w in the goal, inserts $w \geq 0$ and $w < 2^n$
- solves using `arith_tac`, an Isabelle tactic for linear arithmetic

Similar tactic for `sint`: solved test for signed overflow: to prove that, in signed n-bit arithmetic, the addition $x + y$ overflows, that is, `sint x + sint y` $\neq$ `sint (x+y)`, iff the C language term `(((x+y)∧x) & ((x+y)∧y)) >> (n - 1)` is non-zero.

Types containing information about word length

For example, `len_of TYPE(tb t1 t0 t1 t1 t1)` = 23 because `t1 t0 t1 t1 t1` translates to the binary number 10111, ie, 23

```
"len_of TYPE(tb) = 0"
"len_of TYPE('a :: len t0) = 2 * len_of TYPE('a)"
"len_of TYPE('a :: len0 t1) = 2 * len_of TYPE('a) + 1"
```

We use the type class mechanism to prevent use of the type `tb t0` (corresponding to a binary number with a redundant leading zero)

Can also specify the word length and generate the type automatically. For goal `"len_of TYPE(?'a :: len0) = 23"`, this instantiates the variable type `?'a` to `tb t1 t0 t1 t1 t1`

Brian Huffman of Galois Connections has developed types in a similar way, and syntax translation so that the length can be entered or printed out as part of the type.

Length-dependent exhaust theorems

`goal ((x :: word4) >> 2) || (y >> 2) = (x || y) >> 2`

We could prove this by expanding
`x = Pls BIT xa BIT xb BIT xc BIT xd`
(similarly `y`) and calculating both sides by simplification

To enable this we generate a theorem for each word length, eg

```
"[| !!b ba bb bc bd.
w = number_of (Pls BIT b BIT ba BIT bb BIT bc) ==> P;
size w = 4 |] ==> P"
```

also theorems to express a word as a list of bits; eg, for `x` of length 4, expressing `to_bl x` as `[xd, xc, xb, xa]`