Isabelle Theories for Machine Words

Jeremy Dawson[1],[2]

Logic and Computation Program, NICTA
and
Automated Reasoning Group,
Australian National University,
Canberra, ACT 0200, Australia

Abstract

We describe a collection of Isabelle theories which facilitate reasoning about machine words. For each possible word length, the words of that length form a type, and most of our work consists of generic theorems which can be applied to any such type. We develop the relationships between these words and integers (signed and unsigned), lists of booleans and functions from index to value, noting how these relationships are similar to those between an abstract type and its representing set. We discuss how we used Isabelle's `bin` type, before and after it was changed from a datatype to an abstract type, and the techniques we used to retain, as nearly as possible, the convenience of primitive recursive definitions. We describe other useful techniques, such as encoding the word length in the type.

Keywords: machine words, twos-complement, mechanised reasoning

1 Introduction

In formally verifying machine hardware, we need to be able to 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.

Wai Wong [8] developed HOL theories in which words are represented as lists of bits. The type is the set of all words of any length; words of a given length form a subset. Some theorems have the word length as an explicit condition. The theories include some bit-wise operations but not the arithmetic operations.

In [4] Fox descibes HOL theories modelling the architecture of the ARM instruction set. There, the HOL datatype `w32 = W32 of num` is used, that is, the machine word type is isomorphic to the naturals, and the expression `W32` n is to mean the

[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.

[2] `http://users.rsise.anu.edu.au/~jeremy/`

word with unsigned value $n \bmod 2^{32}$. In this approach, equality of machine words does not correspond to equality of their representations.

In [1] Akbarpour, Tahar & Dekdouk describe the formalisation in HOL of fixed point quantities, where a single type is used, and the quantities contain fields showing how many bits appear before and after the point. Their focus is on the approximate representation of floating point quantities.

In [5] Harrison describes the problem of encoding vectors of any dimension n of elements of type A (e.g. reals, or bits) in the type system of HOL, the problem being that a type cannot be parameterised over the value n. His solution is to use the function space type $N \to A$, where N is a *type* which has exactly n values. He discusses the problem that an arbitrary type N may in fact have infinitely many values, when infinite dimensional vectors are not wanted.

In the bitvector library [2] for PVS, which has a more powerful type system, a bit-vector is defined as a function from $\{0, \ldots, N-1\}$ to the booleans. It provides interpretations of a bit-vector as unsigned or signed integers, with relevant theorems.

In this paper we describe theories for Isabelle/HOL [6], for reasoning about machine words. We developed these for NICTA's L4.verified project [7], which aims to provide a mathematical, machine-checked proof of the conformance of the L4 microkernel to a high level, formal description of its expected behaviour. As in [5], each type of words in our formalization is of a particular length. In this work we relate our word types both to the integers modulo 2^n and to lists of booleans; thus we have access to large bodies of results about both arithmetic and logical (bitwise) operations. We have defined all the operations referred to in [4], and describe several other techniques and classes of theorems.

Our theories have been modified recently due to our collaboration with the company Galois Connections, who have developed similar, though less extensive, theories. The Galois theories, though mostly intended to be used for n-bit machine words, are based on an abstract type of integers modulo m (where, for machine words, $m = 2^n$). Thus, in combining the theories (since doing the work described here), we used the more general Galois definition of the abstract type α `word`; our theorems apply when α belongs to an axiomatic type class for which $m = 2^n$.

In this paper we focus on the techniques used to define the machine word type. We defined numerous operations on words which are not discussed here, such as concatenating, splitting, rotating and shifting words. Some of these are mentioned in the Appendix. The Isabelle code files are available at [3].

2 Description of the word-n theories

2.1 The `bin` and `obin` types

Isabelle's `bin` type explicitly represents bit strings, and is important because

- it is used for encoding literal numbers, and an integer entered in an Isabelle expression is converted to a `bin`, thus `read "3"` gives
  `number_of (Pls BIT bit.B1 BIT bit.B1 :: bin)`
  (where x `::` T means that x is of type T) ;
- there is much built-in numeric simplification for numbers expressed as `bin`s, for

example for negation, addition and multiplication, using rules which reflect the usual definitions of these operations for twos-complement integers.

Isabelle changed during development of our theories. Formerly the `bin` type was a datatype, with 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)

Subsequently, in Isabelle 2005, Isabelle's `bin` type changed. The new `bin` type in Isabelle 2005 is an abstract type, isomorphic to the set of all integers, with abstraction and representation functions `Abs_Bin` and `Rep_Bin`.

We found that each of these ways of formulating the `bin` type has certain advantages. We proceed to discuss these, and how we overcame the disadvantages of the new way of defining `bins`. We first describe using the datatype-based definition.

Since at one stage in the course of adapting to this change we were using both the old and new definition of `bins` and associated theorems, we used new names for the old definition, with '`o`' or '`O`' prepended: thus we had the contructors `oPls, oMin, OBIT`, for the datatype `obin`. (We also kept the old function `number_of`, renaming it `onum_of`). So in describing our use of `bins` as formerly defined, we use these names. [3]

2.2 *Definitions using the `obin` datatype*

As these definitions have since been removed, this section is not relevant for using these theories currently. But we give this description to indicate the advantages and disadvantages of the `obin` type, i.e., the former, datatype-based definition of the `bin` type. In fact for some time we continued to use the `obin` type because it is defined as a datatype: only a datatype permits the primitive and general recursive definitions described below.

Using the `obin` datatype allows us to define functions in the most natural way in terms of their action on bits. For example, to define bit-wise complementation, we just used the following primitive recursive definitions:

```
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)"
```

We mention that, apart from the obvious benefit of using a simple definition, it is easier to be sure that it accurately represents the action of hardware that we intend to describe: this is important in theories to be used in formal verification.

Defining bit-wise conjunction using primitive recursion on either of two arguments is conceptually similar, though the expression is not so simple. [4]

[3] More recently, the `bin` type changed again, in development versions of Isabelle during 2006, to be identical to the integers rather than an isomorphic type. So we will omit the functions `Abs_Bin` and `Rep_Bin`, and now our references to the type `bin` indicate an integer expressed using `Pls`, `Min` and `BIT`.

[4] In Isabelle a set of primitive recursive definitions must be based on the cases of exactly one curried argument. It can be easier to use Isabelle's `recdef` package.

We also made considerable use of functions `obin_last` and `obin_rest`, which give the last bit and the remainder, respectively. Again, we defined these functions by primitive recursion using the fact that `obin` is a datatype (the rules correspond to the simplifications *proved* for `bin_last` and `bin_rest`, see §2.3).

In working with the `obin` type, we needed to define the concept of a normalized `obin`, where the combination `oPls OBIT False` does not appear, since it denotes the same sequence of bits, and so the same integer, as `oPls`. So we normalise an `obin` by changing `oPls OBIT False` to `oPls`, and likewise `oMin OBIT True` to `oMin`. Thus the set of normalised `obins` is isomorphic to the set of integers, via the usual twos-complement representation (see theorems `td_int_obin` in §2.5, and `td_ext_int_obin` in §2.6). The following functions relate to normalising `obins`.

```
  mk_norm_obin :: "obin => obin"
  is_norm_obin :: "obin => bool"
```

While use of the `obin` type has the advantage over the `bin` type of being a datatype, the need to prove a large number of lemmas concerning normalisation of `obins` was a significant disadvantage.

2.3 Definitions involving the `bin` type

Our initial development developed words of length n from the set of `obins`. So, for example, we defined the bit-wise complement of a word using `obin_not`, described above, and the addition of two words using addition of `obins`, based on functions to do numerical arithmetic from the Isabelle source files.

However we found the need to deal with words entered literally: `6 :: 'a word` is read as `number_of (Pls BIT bit.B1 BIT bit.B1 BIT bit.B0)`. To simplify `6 && 5 :: 'a word` (where `&&` is our notation for bit-wise conjunction), we found it convenient to use simplifications based on the `bin` type: that is, we wanted to use a function `bin_and`, for bit-wise conjunction of `bins`, rather than `obin_and`. Similarly, dealing with words of length 3, say, we wanted to simplify `11 :: 'a word` to `3` using a function which truncates `bins`, not `obins`.

Since `bin` is not a datatype, we could not define functions on `bins` in the same direct way as on `obins`. So, originally, we defined such functions on `bins` by reference to the corresponding functions on `obins`. To do this we used the functions `onum_of` and `int_to_obin`, which relate the `int` (isomorphic to `bin`) and `obin` types.

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

We had obtained a large number of simplification theorems involving `obins`. Using this approach, we then had to do some rather complex programming to transfer all these simplification theorems, *en masse*, from `obins` to `bins`, so as to avoid proving them all again individually. In this way the parallel use of `obins` and `bins` produced significant extra complexity.

In short, we found that, although the fact of `obin` being a datatype permits simple recursive definitions, the machinery needed to take these definitions and resulting theorems on `obins` and produce definitions and theorems for corresponding functions involving `bins` was unpleasantly cumbersome.

Therefore we examined alternative ways of defining functions in terms of the bit-representation of a bin. First we considered what properties of the bin type resemble the properties of a datatype. The properties of a datatype are:

(a) Different constructors give distinct values

(b) Each constructor is injective (in each of its arguments)

(c) All values of the type are obtained using the constructors

Now we can consider the bin type with "pseudo-constructors" Pls, Min and Bit (where Bit w b is printed and may be entered as w BIT b).

In terms of these "pseudo-constructors" the properties (b) and (c) above hold: in fact property (c) holds using the "pseudo-constructor" Bit alone.

Thus we have these theorems; bin_exhaust enables us to express any bin appearing in a proof as w BIT b. Here !! is Isabelle notation for the universal quantification provided in the meta-logic.

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

Then we can define functions bin_rl, and thence bin_last and bin_rest:

```
bin_rl_def : "bin_rl w == SOME (r, l). w = r BIT l"
bin_rest_def : "bin_rest w == fst (bin_rl w)"
bin_last_def : "bin_last w == snd (bin_rl w)"
```

The SOME function is (partially) defined in Isabelle, by the axiomatic specification "P w ==> P (SOME x. P x)", so its effect here is that if there is a unique choice of r and l to satisfy w = r BIT l, then bin_rl (r BIT l) = (r, l). In fact property (b) gives this uniqueness, and so from that the expected simplification rules bin_last_simps and bin_rest_simps' follow. We then used the numerical characterisation of the BIT operator (effectively, w BIT $b = 2w + b$) to obtain the numerical characterisations of these functions as bin_last_mod and bin_rest_div.

```
bin_last_simps = "bin_last Pls = bit.B0 &
  bin_last Min = bit.B1 & bin_last (w BIT b) = b"
bin_rest_simps' = "bin_rest Pls = Pls &
  bin_rest Min = Min & bin_rest (w BIT b) = w"

bin_last_mod = "bin_last w == if w mod 2 = 0 then bit.B0 else bit.B1"
bin_rest_div = "bin_rest w == w div 2"
```

We also derived a theorem for proofs by induction involving bins.

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

Both bin_exhaust and bin_induct were frequently used in proofs, and they usually made proofs for bins just as easy as the corresponding proofs for obins. Often the theorems and proofs were simpler for bins, e.g.

```
bin_add_not = "x + bin_not x = Min"
obin_add_not = "mk_norm_obin (obin_add x (obin_not x)) = oMin"
```

However obtaining a near-equivalent, for bins, of primitive recursive definitions in obins, was a little more intricate. We have already described the definition of bin_last and bin_rest, and the derivation of simplification rules corresponding to the definitions of obin_last and obin_rest.

Typically a function f defined by primitive recursion would, if bin were a datatype with its three constructors, be defined by giving values vp and vn for f Pls and f Min, and a function fr, where f (w BIT b) is given by fr w b (f w). (The form of the recursion function returned by define_type in the HOL theorem prover makes this explicit). So, using Isabelle's generic mechanism for defining recursive functions, we defined a function bin_rec which, given vp, vn and fr returns a function f satisfying the three equalities shown, but the last only where w BIT b does not equal Pls or Min.

```
bin_rec :: "'a => 'a => (int => bit => 'a => 'a) => int => 'a"

f Pls = vp
f Min = vn
f (w BIT b) = fr w b (f w)
```

In the usual case, we can then prove that this last equation in fact holds for *all* w and b, as we want for a convenient simplification rule. See examples in [3, BinGeneral.thy]. Here are bin_not and bin_and defined in this way:

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

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

Proving these was virtually automatic for bin_not (with one argument), and fairly straightforward for bin_and (with two arguments): see examples in [3, BinGeneral.thy]. This was much easier than maintaining collections of corresponding theorems for the separate types bin *and* obin.

2.4 The type of fixed-length words of given length

As a preliminary step, we define functions which create n-bit quantities. We called these "truncation" functions, although they also lengthen shorter quantities. Both functions will cut down a longer quantity to the desired length, by deleting high-order bits. For an argument shorter than desired, unsigned truncation extends it to the left with zeroes, whereas signed truncation extends it with its most significant bit. Thus bintrunc n w gives Pls followed by n bits, whereas sbintrunc (n-1) w (used for fixed-length words of length n) gives Pls or Min followed by $n - 1$ bits (so here the Pls or Min, is treated as a sign bit, as one of the n bits). We defined bintrunc by primitive recursion on the first argument (the number of bits required) and auxiliary functions bin_last and bin_rest, and sbintrunc similarly.

```
bintrunc, sbintrunc :: "nat => bin => bin"

primrec
  Z : "bintrunc 0 bin = Pls"
  Suc : "bintrunc (Suc n) bin =
      bintrunc n (bin_rest bin) BIT (bin_last bin)"
```

Now we need to set up a type in which the length of words is implicit. The type system of Isabelle is similar to that of HOL in that dependent types are not allowed, so we cannot directly set up a type which consists of (for example) lists of length n. Our solution was that the type of words of length n is α word parametrised over the type α where the word length can be deduced from the type α. As noted, Harrison did this by letting the word length be the number of values of the type α.

We use len_of TYPE(α) for the word length. TYPE(α) is a polymorphic value, of type α itself, whose purpose is essentially to encapsulate a type as a term. [5] In the output of TYPE(α) the type α is printed, which was useful. The function len_of is declared, with polymorphic type (α, printed as 'a, being a type variable) in the library files as shown below. The library files provide the axiom word_size which gives the general formula for the length of a word, but the user must define the value of len_of TYPE(α) for each specific choice of α.

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

A type of fixed-length words is 'a :: len0 word, where len0 is a type class whose only relevance is that it admits a function len_of, and the word length of any w :: 'a :: len0 word is given by the axiom word_size. For each desired word length, the user declares a type (say a), in the class len0, and defines the value len_of TYPE (a) to be the chosen word length. This provides a type of words of that given length.

(Isabelle notation may be confusing here: in w :: 'a :: len0 word, w is a term, 'a is a type variable, len0 is the type class to which 'a belongs, and word is a type constructor. Thus the implicit bracketing is w :: (('a :: len0) word).)

An Isabelle type definition defines a new type whose set of values is isomorphic to a given set. To define each word type we used the definition:

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

where uword_len (len_of TYPE ('a)) is the set of integers, truncated to length n using the function bintrunc described earlier.

The type class len is a subclass of len0, defined by the additional requirement that the word length n is non-zero.

[5] It is used, for example, to express the assertion that a type belongs to a particular type class.

```
len_gt_0 = "0 < len_of TYPE('a::len)"
```

Results involving a signed interpretation of words are limited to this case (naturally, as the word needs to contain a sign bit). [6] Thus the fixed-length word type is abstract, representing a sequence of bits, but such words can be interpreted as unsigned or signed integers. Although the abstract type is defined to be isormorphic to `range (bintrunc n)`, it can be viewed as isomorphic to several different sets. So the set of words of length n is isomorphic to each of the following, with the relevant "type definition theorems" (explained later) given in brackets:

- the set of integers in the range $0 \ldots 2^n - 1$ (`td_uint`)
- the set of integers in the range $-2^{n-1} \ldots 2^{n-1} - 1$ (`td_sint`)
- the set of naturals up to $2^n - 1$ (`td_unat`)
- the set of lists of booleans of length n (`td_bl`)
- the set of functions f of type `nat -> bool` satisfying the requirement that for $i \geq n$, $f\ i = \texttt{False}$ (`td_nth`)

That the type of a word implies its length had some curious consequences. For functions such as `ucast`, which casts a word from one length to another, or `word_rsplit`, which splits a word into a list of words of some given (usually shorter) length, the length of the resulting words is implicit in the result type of the function, not given as an argument. Therefore we get theorems such as `"ucast w = w"` and `"word_rsplit w = [w]"`, where the repeated use of the variable `w` implies that the result word(s) are of the same length as the argument.

2.5 Pseudo type definition theorems

In Isabelle, defining a new type α from a set $S : \rho\ set$ causes the creation of an abstraction function $\texttt{Abs} : \rho \to \alpha$ and a representation function $\texttt{Rep} : \alpha \to \rho$, such that `Abs` and `Rep` are mutually inverse bijections between S and the set of all values of type α. Note that the domain of `Abs` is the type ρ, but that nothing is said about the values it takes outside S. The predicate `type_definition` expresses these properties, and a theorem, `type_definition_`α, stating `type_definition Rep Abs` S, is created for the new type α.

We can use the predicate `type_definition` to express the isormophisms between the set of n-bit words and the other sets mentioned above; we have proved the following "type definition theorems".

```
td_int_obin = "type_definition int_to_obin onum_of (range mk_norm_obin)"
td_uint = "type_definition uint word_of_int (uints (len_of TYPE('a)))"
td_sint = "type_definition sint word_of_int (sints (len_of TYPE('a)))"
td_unat = "type_definition unat of_nat (unats (len_of TYPE('a)))"
td_bl = "type_definition to_bl of_bl
  {bl::bool list. length bl = len_of TYPE('a)}"
td_nth = "type_definition word_nth of_nth
  {f::nat => bool. ALL i::nat. f i --> i < len_of TYPE('a)}"
```

[6] Note that some other results are limited to $n > 0$ because their proof uses theorems from the Isabelle library which apply only in a type class where 0 and 1 are distinct.

These use the following functions between the various types (`of_nat` and `onum_of` have more general types, but are used with these types in these theorems):

```
int_to_obin :: "int => obin"
onum_of :: "obin => int"
word_of_int :: "int => 'a :: len0 word"
uint :: "'a :: len0 word => int"
sint :: "'a :: len word => int"
of_nat :: "nat => 'a :: len0 word"
unat :: "'a :: len0 word => nat"
of_bl :: "bool list => 'a word"
to_bl :: "'a word => bool list"
of_nth :: "(nat => bool) => 'a word"
word_nth :: "'a word => nat => bool"
```

The following define the representing sets referred to above, or were subsequently proved about them:

```
"uints n == range (bintrunc n)"
"sints n == range (sbintrunc (n - 1))"
"unats n == {i. i < 2 ^ n}"
"uints n == {i. 0 <= i & i < 2 ^ n}"
"sints n == {i. - (2 ^ (n - 1)) <= i & i < 2 ^ (n - 1)}"
```

2.6 Extended type definition theorems

As noted, however, these type definition theorems do not say anything about the action of `Abs` outside the set S. But in fact we have defined the abstraction functions to behave "sensibly" outside S, and it is useful to do so. For example, `word_of_int`, which turns an integer in the range $0 \ldots 2^n - 1$ into a word, is defined so that it also behaves "sensibly" on other integers — it takes i and i' to the same word iff $i \equiv i' \pmod{2^n}$. This allows us to use the same abstraction function `word_of_int` in both theorems `td_uint` and `td_sint`.

```
"word_of_int (b mod 2 ^ len_of TYPE('a)) = word_of_int b"
```

The "sensible" definition of `word_of_int` has useful consequences. For example, when we *define* addition of words by `word_add_wi`, where `u` and `v` are words of the same length (and this definition does not involve the addition of `bin`s which are not representatives of words), we also can *prove* the result `wi_hom_add` where `a` and `b` can be *any* integers, whether or not they are values which represent words.

```
word_add_wi : "u + v == word_of_int (uint u + uint v)"
wi_hom_add = "word_of_int a + word_of_int b = word_of_int (a + b)"
```

The following theorems, of the form `Rep` (`Abs` x) $= f\ x$, describe the behaviour of `Abs` outside the representing set S. (It follows that *range* $f = S$).

```
obin_int_obin = "int_to_obin (onum_of n) = mk_norm_obin n"
int_word_uint = "uint (word_of_int a) = a mod 2 ^ len_of TYPE('a)"
unat_of_nat = "unat (of_nat (n::nat)) = n mod 2 ^ len_of TYPE('a)"
```

We therefore defined an extended type definition predicate, as follows:

```
"td_ext Rep Abs A norm ==
  type_definition Rep Abs A & (ALL y. Rep (Abs y) = norm y)"
```

and we have extended type definition theorems including the following:

```
td_ext_int_obin = "td_ext int_to_obin onum_of
  (Collect is_norm_obin) mk_norm_obin"
td_ext_ubin = "td_ext uint word_of_int (uints (len_of TYPE('a)))
  (bintrunc (len_of TYPE('a)))"
td_ext_sbin = "td_ext sint word_of_int (sints (len_of TYPE('a)))
  (sbintrunc (len_of TYPE('a) - 1))"
td_ext_uint = "td_ext uint word_of_int (uints (len_of TYPE('a)))
  (%i. i mod 2 ^ len_of TYPE('a))"
td_ext_unat = "td_ext unat of_nat (unats (len_of TYPE('a)))
  (%i. i mod 2 ^ len_of TYPE('a))"
```

Since $\mathtt{Abs}\ (\mathtt{Rep}\ x) = x$ it follows that $\mathtt{norm} \circ \mathtt{norm} = \mathtt{norm}$, so we call it a normalisation function; we say x is normal if $x = \mathtt{norm}\ y$ for some y, equivalently if $x = \mathtt{norm}\ x$. We also have $\mathtt{norm} \circ \mathtt{Rep} = \mathtt{Rep}$, and $\mathtt{Abs} \circ \mathtt{norm} = \mathtt{Abs}$.

As we frequently had to transfer results about a function on one type to a corresponding function on another type we formalised some general relevant results. Consider a function $f : \rho \to \rho$, where ρ is the representing type in a type definition theorem with normalisation function `norm`. We say x and y are `norm`-equiv[alent] to mean $\mathtt{norm}\ x = \mathtt{norm}\ y$. Then some or all of the following identities may hold:

$\mathtt{norm} \circ f \circ \mathtt{norm} = \mathtt{norm} \circ f$	f takes `norm`-equiv arguments to `norm`-equiv results
$\mathtt{norm} \circ f \circ \mathtt{norm} = f \circ \mathtt{norm}$	f takes normal arguments to normal results
$\mathtt{norm} \circ f = f \circ \mathtt{norm}$	both of the above
$f \circ \mathtt{norm} = f$	f takes `norm`-equiv arguments to the same result
$\mathtt{norm} \circ f = f$	f takes every argument to a normal result

Consider functions $f : \rho \to \rho$ and function $h : \alpha \to \alpha$, where ρ and α are the representing and abstract types in a type definition theorem. These can be related in any of the following ways.

$$h = \mathtt{Abs} \circ f \circ \mathtt{Rep} \tag{1}$$

$$\mathtt{Rep} \circ h = f \circ \mathtt{Rep} \tag{2}$$

$$h \circ \mathtt{Abs} = \mathtt{Abs} \circ f \tag{3}$$

$$\mathtt{Rep} \circ h \circ \mathtt{Abs} = f \tag{4}$$

Of these, (1) would be the typical way to define h in terms of f, and (4) provides the most useful properties, as it implies all the rest; they all imply (1). As for the inverse implications, we obtained a number of general results showing when they are available, depending on which of the properties about `norm` and f above are satisfied (see [3, `TdThs.thy`]). For example, where `norm` is $\mathtt{bintrunc}\ n$, truncation of a `bin` to n bits, and f is addition (with *two* arguments), then f takes `norm`-equiv arguments to `norm`-equiv results. This is the key to obtaining the result `wi_hom_add` shown earlier, of the form of (3) above, from the definition `word_add_wi`, of the form of (1). A similar situation applied in deriving `word_no_log_defs` (see §2.7).

On the other hand, if f is `bin_rest`, or division by 2, and h is `ushiftr1`, unsigned one-bit shift right, then f takes normal arguments to normal results, and when `ushiftr1` is defined from `bin_rest` by (1), equality (2) also holds.

Each type definition theorem is used by the Standard ML (SML) functors `TdThms` or `TdExtThms` to generate a number of consequences, which are collected in structures such as `word` and `int_obin`.

```
structure word = TdThms (struct ... type_definition_word ... end) ;
structure int_obin = TdExtThms (struct ... td_ext_int_obin ... end) ;
```

We note in particular `word_nth.Rep_eqD` and `word_eqI`, derived from it; `word_nth` selects the nth bit of a word, and is written infix as `!!`.

```
word_nth.Rep_eqD = "word_nth x = word_nth y ==> x = y"
word_eqI = "(!!n. n < size u ==> u !! n = v !! n) ==> u = v"
```

The latter was frequently useful in deriving equalities of words. For example, our function `word_cat` concatenates words. We had proved a theorem `word_nth_cat` which gives an expression for `word_cat a b !! n`. Using results like these we could prove two words equal by starting with `word_eqI`, and simplifying. This approach was often useful for proving identities involving concatenating, splitting, rotating or shifting words.

In the same way, the theorem `bin_nth_lem` was useful for proving equality of bins, where `bin_nth x n` is bit `n` of `x`, using theorems such as `nth_bintr`.

```
bin_nth_lem = "bin_nth x = bin_nth y ==> x = y"
nth_bintr = "bin_nth (bintrunc m w) n = (n < m & bin_nth w n)"
```

2.7 Simplifications, `number_of`, literal numbers

As noted ealier, the type `bin` is used in connexion with the function `number_of :: bin => 'a::number` to express literal numbers. When a number (say 5) is entered, it is syntax-translated to `number_of (Pls BIT B1 BIT B0 BIT B1)`. The function `number_of` is defined variously for various types and classes, e.g.:

```
int_number_of_alt = "number_of (w::int) :: int == w"
word_number_of_def =
  "number_of (w::bin) :: 'a::len0 word == word_of_int w"
```

2.7.1 Simplifications for arithmetic expressions

Certain arithmetic equalities, such as associativity and commutativity of addition and multiplication, and distributivity of multiplication over addition, hold for words. We wrote a function `int2lenw` in Standard ML to generate a number of results for words, in `word_arith_eqs`, from the corresponding results about integers. See the file [3, `WordArith.thy`] for details. From these and other results, we showed that the word type is in many of Isabelle's arithmetical type classes (see [3, `WordClasses.thy`]). Therefore many automatic simplifications for these type classes are available for the word type. Thus, for example `a + b + c = (b + d :: 'a :: len0 word)` is simplified to `a + c = d`.

Isabelle is set up to simplify arithmetic expressions involving literal numbers as `bin`s very effectively, using simplification rules which in effect do binary arithmetic, provided that the type of the numbers is in the class `number_ring`. This is the case for words of positive length; unfortunately this does not work for zero-length words, since Isabelle's `number_ring` class requires $0 \neq 1$. Thus an expression such as `(6 + 5 :: 'a :: len word)` gets simplified to `11` automatically, regardless of the word length, which need not be known. Another standard simplification takes `(6 + 5 :: 'a :: len word) = 7` to `iszero (4 :: 'a :: len word)`.

Further simplification of such expressions, i.e., from `(11 :: word2)` to `3` (where `word2` is a type of words of length 2) and from `iszero (4 :: word2)` to `True` depend on the specific word length. We would want to use a theorem like `num_of_bintr`, but we cannot reverse it to use it as a simplification rule because it would loop. Instead we can simplify using `num_abs_bintr` (which is derived from `num_of_bintr` and `word_number_of_def`).

```
num_of_bintr =
  "number_of (bintrunc (len_of TYPE('a)) (b::bin)) = number_of b"
num_abs_bintr = "number_of (b::bin) = word_of_int (len_of TYPE('a)) b"
```

We then need to simplify the word length definition, using the theorem giving `len_of TYPE('a)` for the specific type, then simplify using `bintrunc_pred_simps`, which simplifies an expression like `bintrunc (number_of bin) (w BIT b)`, and finally apply `word_number_of_def` in the opposite direction.

Given an expression such as `iszero (4 :: word2)`, we *can* use the theorem `iszero_word_no` as a simplification rule, and it doesn't loop because the type of `number_of ...` (the argument of `iszero ( )`) is a `word` on the left-hand side but is an `int` on the right-hand side. We would then simplify using the rule giving the word length and `bintrunc_pred_simps`.

```
iszero_word_no = "iszero (number_of (bin::bin)) =
   iszero (number_of (bintrunc (len_of TYPE('a)) bin))"
```

A further approach to simplifying a literal `word` is to simplify an expresssion such as `uint (11 :: word2)`, which means converting `(11 :: word2)` to the integer in the range `uints 2`, i.e. $0 \ldots 2^n - 1$. We would simplify using `uint_bintrunc`, the rule giving the word length and `bintrunc_pred_simps`.

```
uint_bintrunc = "uint (number_of (bin::bin)) =
   number_of (bintrunc (len_of TYPE('a)) bin)"
```

Note that in `uint_bintrunc` the two instances of `number_of` have result types `word` and `int` respectively. Corresponding theorems are available for the signed interpretation of a word, and to simplify `unat` of a literal.

2.7.2 Simplifications for logical expressions

These are more difficult because we do not have a built-in type class. The definition of the bit-wise operations, and how from the definitions we obtained simplifications such as `bin_not_simps` and `bin_and_Bits`, is described in §2.3.

A literal expression such as `22 && 11` can be simplified first using the (derived) rules `word_no_log_defs` (the actual definitions being `word_log_defs`)

```
word_log_defs = ["u && v ==
  number_of (bin_and (uint u) (uint v))", ...]
word_no_log_defs = ["number_of a && number_of b ==
  number_of (bin_and a b)", ...]
```

and then using the simplifications such as `bin_and_Bits` (`word_no_log_defs` and many rules for bit-wise logical operations on `bin`s are in the default simpset).

We derived counterparts for `bin`s of commonplace logical identities such as associativity and commutativity of conjunction and disjunction, and others such as $(x \wedge y) \vee x = x$. We wrote Standard ML code to use these to generate counterparts of these for words, so that one function, `bin2lenw`, sufficed to generate all the corresponding results, found in `word_bw_simps`, about logical bit-wise operations on words. See the file [3, `WordBitwise.thy`] for details.

2.7.3 Special-purpose simplification tactics

Consider the result (for words) `"(x < x - z) = (x < z)"`: each inequality holds iff calculating $x - z$ causes underflow. Several results required about words, such as this one, could be proved by translating into goals involving sums or differences of integers, together with case analyses as to whether overflow or underflow occurred or not. So we developed tactics for these: `uint_pm_tac` does the following

- unfolds definitions of $\leq$, using `word_le_def` (similarly for $<$)
- unfolds occurrences of `uint (a + b)` using `uint_plus_if'` (similarly for `uint (a - b)`)
- for every occurrence of `uint w` in the goal, inserts `uint_range'`
- solves using `arith_tac`, an Isabelle tactic for solving linear arithmetic

```
word_le_def = "a <= b == uint a <= uint b"
uint_plus_if' = "uint (a + b) =
  (if uint a + uint b < 2 ^ len_of TYPE('a) then uint a + uint b
  else uint a + uint b - 2 ^ len_of TYPE('a))"
uint_range' = "0 <= uint w & uint w < 2 ^ len_of TYPE('a)"
```

This proved effective for a reasonable number of goals that arose in practice; it relies on the fact that `arith_tac` is very effective for goals involving $<$, $<=$, $+$ and $-$ for integers. Details of the code are in [3, `WordArith.thy`].

We developed similar routines for `sint`, which were used to solve a problem posed by a referee: 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.

2.8 Types containing information about word length

We have defined types which contain information about the length of words. For example, `len_of TYPE(tb t1 t0 t1 t1 t1)` $= 23$ because `t1 t0 t1 t1 t1` translates to the binary number 10111, that is, 23. The relevant simplification rules (which are axioms, and so in the default simpset) are

```
len_tb : "len_of TYPE (tb) = 0"
len_t0 : "len_of TYPE ('a :: len t0) = 2 * len_of TYPE ('a)"
len_t1 : "len_of TYPE ('a :: len0 t1) = 2 * len_of TYPE ('a) + 1"
```

and so `len_of TYPE(tb t1 t0 t1 t1 t1)` is simplified to 23 automatically.

We use the type class mechanism to prevent use of the type `tb t0` (corresponding to a binary number with a redundant leading zero); the class `len` is used for words whose length is non-zero and we used the arity declarations shown, although the instance declarations shown are then deducible.

```
arities tb :: len0
arities t0 :: (len) len0          instance t0 :: (len) len
arities t1 :: (len0) len0         instance t1 :: (len0) len
```

By the `arities` declaration for `t0`, we can make use of a type α `t0` only where α is in the class `len` (indicating a non-zero word length), which prevents using `tb` as α. The deduced `instance` results mean that any type α `t1` is of class `len`, and likewise for α `t0`, when α is of class `len`.

It is also possible to specify the word length rather than the type, and have the type generated automatically. For example, for a goal with a variable type, e.g. `"len_of TYPE(?'a :: len0) = 23"`, repeated use of appropriate introduction rules (`len_no_intros`) will instantiate the variable type `?'a` to `tb t1 t0 t1 t1 t1`.

See [3, `Autotypes.thy`] for details, and for further relevant theorems. 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.

2.9 Length-dependent exhaust theorems

Consider the goal `"((x :: word6) >> 2) || (y >> 2) = (x || y) >> 2"` where `x >> 2` means `x`, with bits shifted two places to the right, and `x || y` is bit-wise disjunction. We could prove such a theorem by expanding `x` by

```
x = Pls BIT xa BIT xb BIT xc BIT xd BIT xe BIT xf
```

(similarly `y`) and calculating both sides by simplification. To enable this we generate a theorem for each word length; the one for word length 6 is shown.

```
"[| !!b ba bb bc bd be.  w = number_of
  (Pls BIT b BIT ba BIT bb BIT bc BIT bd BIT be) ==> P;
  size w = 6 |] ==> P"
```

We also generated theorems to express a word as a list of bits; for example, for `x` of length 6, expressing `to_bl x` as `[xf, xe, xd, xc, xb, xa]`.

Such a theorem can then be instantiated; for example, for the goal above, one would use the theorem for word length 6 twice, instantiating it with `x` and `y` respectively. An example is in [3, `Word32.ML`].

We are also developing techniques for translating a goal into a format suitable for handing over to a SAT solver. This involves expressing a word of length n as a sequence of n bits, and we have used these theorems for this purpose also.

3 Conclusion

The theories we describe have been used extensively in the NICTA's L4.verified project, which requires reasoning about the properties of machine words and their operations. We have discussed how we defined types of words of various lengths, with theorems which apply to words of any length. We have shown how to make definitions about `bins` by a procedure sufficiently resembling primitive recursion to be practical and useful. We have taken advantage of the fact that the set of words is isomorphic to several different sets and used "pseudo" type definition theorems to use these and derive relevant results in an efficient and uniform way. Finally we described other useful techniques, such as how to create types which automatically imply the word length, using type constructors corresponding to binary digits.

In these theories, where a single type of words has a definite length, definitions and theorems about joining or splitting words were difficult. In this area, using the bit-vector library of PVS [2], with its more powerful type system, might be easier.

A noteworthy feature of the work was the value of Standard ML as the user interface language. As described in §2.6 we used its structures and functors, which were very convenient for generating a large number of theorems of the same pattern without repeating code. We used its capabilities as a programming language to write a number of functions for generating theorems *en masse*, such as the SML function `int2lenw` and `bin2lenw` which were used to generate respectively 15 and 31 theorems about words from corresponding theorems about `ints` and `bins`. Coding in SML was also indispensible for the simplification procedures used to provide automatic simplification of literal expressions, for tactics such as `uint_pm_tac`, for generating the theorems of §2.9 for arbitrary n, and for HOL-style conversions, which we used in the proofs. Of course, more mundane uses of its capabilities, such as applying a transformation to a list of theorems, was commonplace in our work.

3.0.1 Acknowledgements

I wish to thank Gerwin Klein and anonymous referees for helpful suggestions, and John Matthews for the contribution of Galois Connections to the present code.

References

[1] Behzad Akbarpour, Sofiène Tahar & Abdelkader Dekdouk. Formalization of Fixed-Point Arithmetic in HOL. Formal Methods in System Design 27, 173–200, 2005.

[2] Ricky W Butler, Paul S Miner, Mandayam K Srivas, Dave A Greve, Steven P Miller. A New Bitvectors Library For PVS. NASA, Langley, USA, 1997.

[3] Jeremy Dawson. Isabelle `word` theory files. NICTA. http://users.rsise.anu.edu.au/~jeremy/isabelle/l4/

[4] Anthony Fox. A HOL Specification of the ARM Instruction Set Architecture. Computer Laboratory, University of Cambridge, 2001.

[5] John Harrison. A HOL Theory of Euclidean Space. In Theorem Proving in Higher Order Logics, (TPHOLs 2005). Lecture Notes in Computer Science 3603, 114-129.

[6] L C Paulson, T Nipkow. Isabelle. http://isabelle.in.tum.de/

[7] L4.verified project, NICTA. http://ertos.nicta.com.au/research/l4.verified/

[8] Wai Wong. Modelling Bit Vectors in HOL: the word library. In Higher Order Logic Theorem Proving and its Applications (HUG '93), Lecture Notes in Computer Science 780, 371-384.