

Electronic Notes in Theoretical Computer Science 250 (2009) 55–70

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Isabelle Theories for Machine Words

Jeremy Dawson[1](#_bookmark0)

*Logic and Computation Group, School of Computer Science, 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

# Introduction

In formally verifying machine hardware, we need to be able to deal with the prop- erties 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](#_bookmark18)] 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](#_bookmark15)] Fox descibes HOL theories modelling the architecture of the ARM instruc- tion 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 word with unsigned value *n* mod 232. In this approach, equality of machine words does not correspond to equality of their representations.

1 <http://users.rsise.anu.edu.au/~jeremy/>

1571-0661 © 2009 Elsevier B.V. Open access under [CC BY-NC-ND license](http://creativecommons.org/licenses/by-nc-nd/3.0/).

doi:10.1016/j.entcs.2009.08.005

In [[1](#_bookmark12)] Akbarpour, Tahar & Dekdouk describe the formalisation in HOL of fixed point quantities, where a single type is used, and the quantities contain fields show- ing how many bits appear before and after the point. Their focus is on the approx- imate representation of floating point quantities.

In [[5](#_bookmark16)] 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 → 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](#_bookmark13)] for PVS, which has a more powerful type system, a bit-vector is defined as a function from *{*0*,...,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](#_bookmark17)], for reasoning about machine words. We developed these for NICTA’s L4.verified project [[7](#_bookmark19)], 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](#_bookmark16)], 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 (bit- wise) operations. We have defined all the operations referred to in [[4](#_bookmark15)], 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](#_bookmark14)].

# Description of the word-n theories

* 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 bins, 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 ad- vantages. 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 contruc- tors 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. [2](#_bookmark1)

* 1. *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 argu-

2 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.

ments is conceptually similar, though the expression is not so simple. [3](#_bookmark3)

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](#_bookmark2)).

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](#_bookmark7), and td\_ext\_int\_obin in *§*[2.6](#_bookmark9)). 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.

* 1. *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

3 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.

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:

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

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)](#_bookmark4) and [(c)](#_bookmark5) above hold: in fact property [(c)](#_bookmark5) holds using the “pseudo-constructor” Bit alone.

Thus we have these theorems; bin\_exhaust enables us to express any bin ap- pearing in a proof as w BIT b. Here !! is Isabelle notation for the universal quan- tification 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)](#_bookmark4) 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* = 2*w* + *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 func- tion 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](#_bookmark14), 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](#_bookmark14), BinGeneral.thy]. This was much easier than maintaining collections of corre- sponding theorems for the separate types bin *and* obin.

* 1. *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. [4](#_bookmark6) 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).)

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

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.

len\_gt\_0 = "0 < len\_of TYPE(’a::len)"

Results involving a signed interpretation of words are limited to this case (nat- urally, as the word needs to contain a sign bit). [5](#_bookmark8) 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 *...* 2*n −* 1 (td\_uint)
* the set of integers in the range *−*2*n−*1 *...* 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 ≥ n*, *f i* = 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.

* 1. *Pseudo type deﬁnition theorems*

In Isabelle, defining a new type *α* from a set *S* : *ρ set* causes the creation of an abstraction function Abs : *ρ → α* and a representation function Rep : *α → ρ*, 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*.

5 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.

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)}"

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)}"

* 1. *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 *...* 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 ≡ i'* (mod 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 bins 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 Abs (Rep *x*) = *x* it follows that norm *◦* norm = norm, so we call it a normalisation function; we say *x* is normal if *x* = norm *y* for some *y*, equivalently if *x* = norm *x*. We also have norm *◦* Rep = Rep, and Abs *◦* norm = 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* : *ρ → ρ*, 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 norm *x* = norm *y*. Then some or all of the following identities may hold:

norm *◦ f ◦* norm = norm *◦ f f* takes norm-equiv arguments to norm-equiv results

norm *◦ f ◦* norm = *f ◦* norm *f* takes normal arguments to normal results

norm *◦ f* = *f ◦* norm both of the above

*f ◦* norm = *f f* takes norm-equiv arguments to the same result

norm *◦ f* = *f f* takes every argument to a normal result

Consider functions *f* : *ρ → ρ* and function *h* : *α → α*, 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* = Abs *◦ f ◦* Rep (1)

Rep *◦ h* = *f ◦* Rep (2)

*h ◦* Abs = Abs *◦ f* (3)

Rep *◦ h ◦* Abs = *f* (4)

Of these, ([1](#_bookmark9)) would be the typical way to define *h* in terms of *f* , and ([4](#_bookmark9)) provides the most useful properties, as it implies all the rest; they all imply ([1](#_bookmark9)). 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](#_bookmark14), TdThs.thy]). For example, where norm is 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](#_bookmark9)) above, from the definition word\_add\_wi, of the form of ([1](#_bookmark9)). A similar situation applied in deriving word\_no\_log\_defs (see *§*[2.7](#_bookmark10)).

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](#_bookmark9)), equality ([2](#_bookmark9)) 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 struc- tures 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 *n*th 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)"

* 1. *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"

* + 1. *Simpliﬁcations for arithmetic expressions*

Certain arithmetic equalities, such as associativity and commutativity of addi- tion and multiplication, and distributivity of multiplication over addition, hold for words. We wrote a function int2lenw in Standard ML to generate a num- ber of results for words, in word\_arith\_eqs, from the corresponding results about integers. See the file [[3](#_bookmark14), WordArith.thy] for details. ¿From these and other results, we showed that the word type is in many of Isabelle’s arithmeti- cal type classes (see [[3](#_bookmark14), WordClasses.thy]). Therefore many automatic simplifi- cations 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 bins 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 */*= 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 *...* 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.

* + 1. *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](#_bookmark2).

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 bins are in the default simpset).

We derived counterparts for bins of commonplace logical identities such as as- sociativity and commutativity of conjunction and disjunction, and others such as (*x ∧ y*) *∨ 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 cor- responding results, found in word\_bw\_simps, about logical bit-wise operations on words. See the file [[3](#_bookmark14), WordBitwise.thy] for details.

* + 1. *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 *≤*, 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](#_bookmark14), WordArith.thy].

We developed similar routines for sint, which were used to solve a prob- lem posed by a referee: to prove that, in signed *n*-bit arithmetic, the addition *x* + *y* overflows, that is, sint x + sint y */*= sint (x+y), iff the C language term (((x+y)^x) & ((x+y)^y)) >> (n - 1) is non-zero.

* 1. *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 trans- lates 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 vari- able type, e.g. "len\_of TYPE(?’a :: len0) = 23", repeated use of appropri- ate introduction rules (len\_no\_intros) will instantiate the variable type ?’a to

tb t1 t0 t1 t1 t1.

See [[3](#_bookmark14), 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.

* 1. *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 respec- tively. An example is in [[3](#_bookmark14), 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.

# 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](#_bookmark13)], 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](#_bookmark9) 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](#_bookmark11) 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.

# Acknowledgement

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, Sofi`ene 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/](http://users.rsise.anu.edu.au/~jeremy/isabelle/l4/) [isabelle/l4/](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.