LCF-style Bit-Blasting in HOL4

Anthony C. J. Fox

Computer Laboratory, University of Cambridge, Cambridge, UK

Abstract. This paper describes a new proof tool for deciding bit-vector problems in HOL4. The approach is based on "bit-blasting", wherein word expressions are mapped into propositional formulas, which are then handed to a SAT solver. Significantly, the implementation uses the LCF approach, which means that the soundness of the tool is guaranteed by the soundness of HOL4's logical kernel.

1 Introduction

Interactive theorem provers are often used in areas such as hardware design, computer architectures, compilers, operating systems, protocols and cryptography. Inevitably these domains provide an abundant source of bit-vector problems, and users would like the ability to prove these goals automatically. For example, consider the formula

$$(a_{(:32)} \;\&\&\; \mathbf{3} = \mathbf{0}) \Rightarrow ((a + \mathbf{4}) \;\&\&\; \mathbf{3} = \mathbf{0})$$

where $a_{(:32)}$ indicates that a is a 32-bit word and $\&\&$ is bitwise-and. This theorem shows that adding four to a word aligned memory address does not break the alignment property. Such goals are potentially challenging for HOL4 users, but this goal can now be solved fully automatically and quickly (0.05 s).

The tool presented here uses an established technique called *bit-blasting*. Although the implementation is much simpler than highly advanced bit-vector decision procedures (such as [3]), the tool is implemented in an LCF style, which is of great advantage with respect to ensuring logical soundness. The principle design objective was to produce a simple tool that can handle many "small but somewhat tricky" bit-vectors problems that often arise during interactive proofs. In this sense the tool has already been very successful. Recently it has been used to great effect by Magnus Myreen during machine code verification as part of the Jitawa project, see [5].

There will be bit-vector problems that are too complex for the tool to handle quickly. Nevertheless, complex problems can often be tackled with some human guidance.[1] As with provers such as PVS and Isabelle, HOL4 users can also call external high-performance proof tools, treating these tools as *oracles*. Recently Tjark Weber has integrated SMT solvers with bit-vector capabilities into HOL, see [2]. Theorems that are tagged as coming from oracles are considered undesirable in HOL, since they do not offer the high assurance of LCF-style proofs. Weber uses our new LCF procedure to safely reconstruct SMT bit-vector proofs.

[1] Users can discover and apply helpful abstractions, case-splits and simplifications.

2 Representation of Bit-vectors

Bit-vectors can be represented as *finite Cartesian products* $\mathbb{B}^n$ where n is the finite, fixed *width* (or length) of the bit-vector. At first glance, it does not seem possible to *directly* represent the set $\mathbb{B}^n$ using HOL4's simple type system. However, in [4] John Harrison showed that parametric polymorphism can be used to specify vector widths. Using Harrison's approach bit-vectors are represented by the type $\mathsf{bool}[\alpha]$ and the word length is given by the term $\dim(:\alpha)$, where α is a type variable. Readers are referred to Harrison's paper for the details of this approach.

Note that HOL4's parser and pretty-printer support *numeric types*, which makes it easy to work with concrete bit-vector instances. For example, 32-bit words are represented by $\mathsf{bool}[32]$ and we have $\dim(:32) = 32$.

This representation of bit-vectors has been used in HOL4 since 2005 and has worked well in practice. In particular we gain the benefits of exploiting HOL4's well established support for parametric polymorphism. There are some limitations — the type system is not intelligent with regard to result type of word extraction and concatenation, but some extra parsing tool support is provided to help address this drawback.

3 Bit-vector Operations

Having defined a representing type for bit-vectors, one can define a collection of standard operations. These generally split into three camps:

1. *Arithmetic operations*: $+$, unary $-$, binary $-$, $\times$, $\div$, mod, $<$, $\leq$, $>$, and $\geq$. In some cases there are signed (2's complement) and unsigned variants.
2. *Bitwise/logical operations*: $\neg$ (bitwise NOT), && (bitwise AND), $\|$ (bitwise OR), $\oplus$ (bitwise XOR), shifts and rotations.
3. *Casting maps*: unsigned maps to and from $\mathbb{N}$ (w2n and n2w), signed maps to and from $\mathbb{Z}$ (w2i and i2w), unsigned and signed word-to-word maps (w2w and sw2sw), and word extraction and concatenation.

These and other operations are defined through the use of a finite Cartesian product *binder* FCP:

$$(\,\mathrm{FCP}\,) : (\mathsf{num} \to \beta) \to \beta[\alpha]$$

and a *projection* function

$$(\,'\,) : \beta[\alpha] \to \mathsf{num} \to \beta\ .$$

For bit-vectors, β is specialised to bool. The FCP binder constructs a Cartesian product from a function.[2] The bitwise operators are easy to define; for example, bitwise conjunction $\&\& : \mathsf{bool}[\alpha] \to \mathsf{bool}[\alpha] \to \mathsf{bool}[\alpha]$ is defined as follows:

$$a \;\&\&\; b =_{def} \mathrm{FCP}\ i.\ (a\ '\ i) \wedge (b\ '\ i)\ .$$

[2] In HOL4, the binder syntax FCP $i.\ f\ i$ is used to denote FCP $(\lambda i.\ f\ i)$.

It is also easy to prove that

$$\forall a, b : \mathsf{bool}[\alpha]\ i : \mathsf{num}.\ \ i < \dim(:\alpha) \Rightarrow ((a\ \&\&\ b)\ '\ i = (a\ '\ i) \wedge (b\ '\ i))\ .$$

The arithmetic operations are defined using the maps w2n and n2w. For example, addition is defined as follows:

$$a + b =_{def} \text{n2w}(\text{w2n}(a) + \text{w2n}(b))\ .$$

Here bit-vector addition is on the left-hand side and natural number addition is on the right-hand side. The natural number mappings are defined as follows:

$$\text{w2n}(a : \mathsf{bool}[\alpha]) =_{def} \sum_{0 \le i < \dim(:\alpha)} \textbf{if}\ a\ '\ i\ \textbf{then}\ 2^i\ \textbf{else}\ 0$$

and

$$\text{n2w}(n) =_{def} \text{FCP}\ i.\ (n \operatorname{div} 2^i) \bmod 2 = 1\ .$$

To facilitate bit-blasting, the following theorems are proved:

$$\vdash \forall x\, y.\ x + y = \text{FCP}\ i.\ \text{Sum}\ i\ (\lambda i.\ x\ '\ i)\ (\lambda i.\ y\ '\ i)\ \bot$$
$$\vdash \forall x\, y.\ x - y = \text{FCP}\ i.\ \text{Sum}\ i\ (\lambda i.\ x\ '\ i)\ (\lambda i.\ \neg(y\ '\ i))\ \top$$
$$\vdash \forall x\, y.\ x_{(:\alpha)} < y = \neg(\text{Carry}\ (\dim(:\alpha))\ (\lambda i.\ x\ '\ i)\ (\lambda i.\ \neg(y\ '\ i))\ \top)$$

where $\top$ represents true, $\bot$ is false, $<$ is *unsigned* less-than (<+ in HOL4) and

$$\text{Carry}, \text{Sum} : \mathsf{num} \to (\mathsf{num} \to \mathsf{bool}) \to (\mathsf{num} \to \mathsf{bool}) \to \mathsf{bool} \to \mathsf{bool}$$

are primitive recursive, circuit-like specifications for a ripple-carry adder. At present bit-blasting of division is not supported and *general* multiplication has been constrained to small word sizes (at the moment less than nine bits). However, multiplication by a constant is supported at all word lengths, e.g. $\mathbf{3} \cdot x$.

4 Bit-blasting

Many common bit-vector problems can be readily solved using simplification, where collections of algebraic equations and rules are applied as rewrites. This approach works especially well when working with bit-vector operations that come exclusively from one category, i.e. all arithmetic or all bitwise. For example, the following are *automatically* proved via standard word simplification:

$$b \cdot \mathbf{2} + \mathbf{4} \cdot a - b + a - b = \mathbf{5} \cdot a,$$
$$a \oplus (b \parallel a \parallel \neg b) = (\neg(a\ \&\&\ \neg a) \parallel a \parallel b)\ \&\&\ \neg a\ .$$

Things get harder when operations are freely mixed, for example:

$$((a_{(:32)} + \text{w2w}(b_{(:8)}))[7:0] = a[7:0] + b,$$
$$\neg(x\ '\ 0) \Rightarrow ((\mathbf{17} \cdot x_{(:8)})\ \&\&\ \mathbf{6} = \mathbf{7} \cdot x)$$

Here, $\text{w2w} : \mathsf{bool}[\alpha] \to \mathsf{bool}[\beta]$ is a word-to-word mapping (this zero extends on expansion and truncates on contraction) and $x[i : j]$ represents extraction over a bit range. These are the sorts of goals that users may find hard to prove manually but that bit-blasting can handle with ease.

Implementation. As with many proof procedures in HOL4, the underlying LCF implementation is in the form of a *conversion*, which converts a term into an equality theorem, where the original term occurs on the left-hand side. A decision procedure succeeds if the right-hand side after conversion is $\top$.

To demonstrate the implementation, the following example formula is used:

$$(a_{(:2)} < b) \wedge (b < c) \wedge (c < d) \Rightarrow (a \parallel (\mathbf{3} \cdot d) = \mathbf{1}) \ .$$

The domain is 2-bit words (i.e. $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$). The chain of orderings in the antecedent imply that a must be the infimum $\mathbf{0}$ and d must be the supremum $\mathbf{3}$ and hence the equality is known to hold (since $\mathbf{0} \parallel (\mathbf{3} \cdot \mathbf{3}) = \mathbf{1}$).

The decision procedure calls the conversion `BBLAST_CONV`, which starts by applying standard bit vector simplifications.[3] The next stage is identifying sub-terms that are amenable to bit-blasting — here there are four such sub-terms:

$$a < b \qquad b < c \qquad c < d \qquad a \parallel (\mathbf{3} \cdot d) = \mathbf{1}$$

The inequalities are expanded as follows:

$$\begin{aligned} a_{(:2)} < b &\mapsto \neg(\text{Carry } 2\ (\lambda i.\ a\ '\ i)\ (\lambda i.\ \neg(b\ '\ i))\ \top) \\ &\mapsto \neg(a\ '\ 1 \wedge \neg b\ '\ 1 \vee (a\ '\ 1 \vee \neg b\ '\ 1) \wedge (a\ '\ 0 \vee \neg b\ '\ 0)) \ . \end{aligned}$$

The equality (consequent) sub-term is more complicated. The multiplication by a constant is eliminated using the conversion $\mathbf{3} \cdot x \mapsto x \ll 1 + x$ where $\ll$ is logical shift-left. The term is then rewritten, introducing the FCP binder wherever possible. For example, we use the definition of shift-left:

$$x_{(:\alpha)} \ll n =_{def} \text{FCP } i.\ i < \dim(:\alpha) \wedge n \leq i \wedge (x\ '\ (i - n)) \ .$$

The result at this stage is of the form:

$$(a \parallel (\mathbf{3} \cdot d) = \mathbf{1}) \mapsto (\text{FCP } i.\ a\ '\ i \vee (\text{FCP } i.\ \text{Sum } i\ \ldots)\ '\ i) = (\text{FCP } i.\ i = 0) \ .$$

In general, word equalities will be converted into the form FCP f = FCP g for some pair of functions $f, g : \mathsf{num} \to \mathsf{bool}$. The next stage is to use the theorem:

$$(\text{FCP } f)_{(:2)} = \text{FCP } g \Leftrightarrow (f(0) = g(0)) \wedge (f(1) = g(1)) \ .$$

We now try to symbolically evaluate f and g at each bit position. Care is taken to perform this evaluation efficiently – there is a preliminary stage that iteratively generates sets of rewrites for Sum and Carry sub-terms.[4] With our example, the evaluation at position zero gives us: $f(0) = a\ '\ 0 \vee d\ '\ 0$ and $g(0) = \top$. If $f(0) = g(0)$ rewrites to false then we can quit early, knowing that the entire

[3] Simplification does not alter our example goal. However, at best simplification will convert terms to $\top$ or $\bot$, in which case bit-blasting will not occur.

[4] Note that Sum terms can be nested, so the order of rewrite generation is important.

word equality is false, but with this example we move on to the next bit position. After some basic Boolean simplification, we have the conversion:

$$(a \parallel (\mathbf{3} \cdot d) = \mathbf{1}) \mapsto (a\ '\ 0 \vee d\ '\ 0) \wedge \neg(a\ '\ 1 \vee (d\ '\ 0 \Leftrightarrow \neg d\ '\ 1)) \ .$$

A SAT solver is called on this proposition, which reveals that it is *contingent*, i.e. we cannot rewrite it to $\top$ or $\bot$. Having converted all of the original sub-terms, we are left with the proposition:

$$\begin{aligned}
&\neg(a\ '\ 1 \wedge \neg b\ '\ 1 \vee (a\ '\ 1 \vee \neg b\ '\ 1) \wedge (a\ '\ 0 \vee \neg b\ '\ 0)) \wedge \\
&\neg(b\ '\ 1 \wedge \neg c\ '\ 1 \vee (b\ '\ 1 \vee \neg c\ '\ 1) \wedge (b\ '\ 0 \vee \neg c\ '\ 0)) \wedge \\
&\neg(c\ '\ 1 \wedge \neg d\ '\ 1 \vee (c\ '\ 1 \vee \neg d\ '\ 1) \wedge (c\ '\ 0 \vee \neg d\ '\ 0)) \Rightarrow \\
&\qquad (a\ '\ 0 \vee d\ '\ 0) \wedge \neg(a\ '\ 1 \vee (d\ '\ 0 \Leftrightarrow \neg d\ '\ 1)) \ .
\end{aligned}$$

This time calling the SAT solver gives us $\top$ and the decision procedure succeeds.

A very useful facility of this SAT based decision procedure is the ability to print counterexamples. For example, calling the procedure on $a \leq a + b_{(:4)}$ gives the counterexample $a \mapsto \mathbf{12}$ and $b \mapsto \mathbf{4}$.

Performance. Unsurprisingly, bit-blasting decision procedures can encounter severe complexity problems. Figure 1 illustrates a worst-case scenario, showing the time required to apply `BBLAST_CONV` to the following sequence of terms:

$$x = a + b, \quad x = a + b + c, \quad x = a + b + c + d, \quad x = a + b + c + d + e \ .$$

The second term took just over four minutes to convert for 128-bit words using a 2.5 GHz Core 2 Duo machine with 4 GB of RAM. The resulting theorem is very large and calling the SAT solver (to no avail) took up 82 s.[5] However, this gives a false impression. In practice, run-times are typically much more respectable. Consider, the following equations:

$$(\mathbf{193} \cdot a)\ \&\&\ \mathbf{7} = \mathbf{7}\ \&\&\ a \tag{1}$$

$$(a\ \&\&\ b) + (a \parallel b) = a + b \tag{2}$$

$$(\mathbf{8} \cdot a + (b\ \&\&\ \mathbf{7})) \ggg 3 = a\ \&\&\ (-\mathbf{1} \ggg \mathbf{3}) \tag{3}$$

where $\ggg$ is logical shift-right. Figure 2 shows the timings for these problems. Equation 1 has two nested additions (from multiplying by **193**) but this time the 128-bit case takes around 45 s to solve. The difference here is that we can rewrite to $\top$ at each bit position, and this avoids passing on a large term to the SAT solver. In Equation 2 we have two additions but this time they are not nested, so the goal is less complex. In Equation 3 we have one addition, with that addition becoming "simple" after bit position three (the bit value of the second argument becomes false). Equation 3, and goals without additions, scale very well to large word sizes. When working with typical *machine* word sizes (i.e. 32-bit and 64-bit words), the run-times for all three problems are in the order of seconds.

[5] The timings do not include the printing of terms. For obvious reasons the maximum print-depth must be limited when working with very large terms.

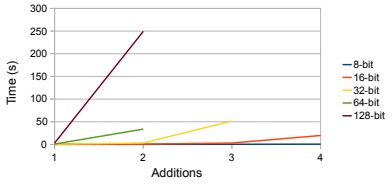

Fig. 1. Nested additions (conversion).

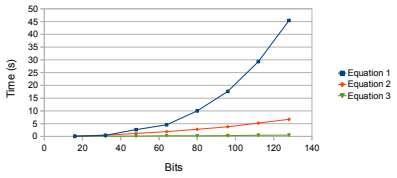

Fig. 2. Typical problems.

5 Summary

This paper has demonstrated that it is possible to implement a useful, practical and efficient LCF-style proof procedure for bit-vectors in HOL4. The source code is distributed with HOL4. This development has been made possible through the work of Hasan Amjad and Tjark Weber in integrating modern SAT solvers (zChaff and MiniSat) into HOL provers using the LCF approach, see [1]. Michael Norrish's DPLL based proof procedure (described in the HOL4 Tutorial, see `hol.sf.net`) is also used to quickly handle small propositions.

There are circumstances when bit-blasting is not appropriate and users will find that traditional methods (simplification and lemma construction) are required — either for efficiency reasons or when proving general results, i.e. for arbitrary word sizes. The proof technique relies on formulating and symbolically evaluating a bit-stream function $f : \mathsf{num} \to \mathsf{bool}$. It is this requirement that ultimately determines the scope of the procedure. For example, goals involving $x[m : n]$, $\mathrm{w2n}(w)$, $\mathrm{w2i}(w)$, $\mathrm{n2w}(n)$ and $\mathrm{i2w}(i)$, where m, n and i are not constant values, will require reasoning over $\mathbb{N}$ or $\mathbb{Z}$. Nevertheless, the tool does have excellent coverage and it automatically handles many common goals that users could easily waste many hours solving manually. The tool can even handle basic existential goals, e.g. it proves $\exists\, x\, y.\ (x \cdot y = \mathbf{12}_{(:8)}) \wedge x < y$ in 2.3 s. One possible area for performance improvements is to automatically introducing abstractions, e.g. replacing "costly but non-critical" sub-expressions with variables.

Many thanks to Mike Gordon, Magnus Myreen and Tjark Weber for helpful feedback and assistance.

References

1. Amjad, H., Weber, T.: Efficiently checking propositional refutations in HOL theorem provers. Journal of Applied Logic 7(1), 26–40 (2007)
2. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 179–194. Springer (2010)
3. Bryant, R.E., Kroening, D., Puaknine, J., Seshia, S.A., Strichman, O., Brandy, B.: Deciding bit-vector arithmetic with abstraction. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 358–372. Springer (2007)
4. Harrison, J.: A HOL theory of Euclidean Space. In: Hurd, J., Melham, T.F. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 114–129. Springer (2005)
5. Myreen, M.O., Davis, J.: A verified runtime for a verified theorem prover. In: ITP 2011. LNCS (to appear), Springer (2011)