Computer Arithmetic: Logic, Calculation, and Rewriting

M. Benini[*]
University of Warwick[†]

D. Nowotka
University of Warwick[‡]

C. Pulley
University of Huddersfield[§]

March 6, 2002

1 Introduction

Computer arithmetic is the mathematical theory which underlies the way calculational machines operate on integer numbers.

Computers manipulate *integer numbers* of a finite, fixed precision, internally represented as strings of bits of fixed length. A processor's hardware [3] is built to perform additions, multiplications, and other standard arithmetical operations along with *logical* operations like "or", "and", "not", "exclusive or", and so on.

The distinguishing features of computer arithmetic are:

- *logical* operations, i.e., the ability to calculate bit per bit the conjunction, disjunction and negation of *integers*. For clarity, since we deal with a logical theory, we will refer to these operations with the adjective *bitwise*;
- fixed precision, which means every representable number lies in a fixed, well-defined range of values and every operation must signal exceptions, when unable to provide a result which fits into that range, usually by means of carry and/or overflows bits.

The goal of this paper is to provide a logical formulation for computer arithmetic, suitable for coding in a theorem proving environment, along with an algorithmic, as opposed to logically declarative, calculational engine

[*] This work has been sponsored by EPSRC under grant GR/K52447.

[†] marcob@dcs.warwick.ac.uk
[‡] dirk@dcs.warwick.ac.uk
[§] C.J.Pulley@hud.ac.uk

based on equational rewriting techniques, which forms the natural counterpart for the logical representation, and a decision procedure providing efficiency in computation, without compromising validity of results.

Even if it may appear to be a very specialized topic, an integrated decision procedure and a calculational engine for computer arithmetic is very general and very useful as well.

Dropping the constraint of finite precision, computer arithmetic is actually an extension of the standard theory of integer numbers [16], i.e., it provides the usual operations on integers as well as bitwise operations.

From another point of view, the one of formal verification [6], computer arithmetic is essential, since it is the theory we are really dealing with when we execute a program, or when we design a digital circuit.

What we are about to show is a logical theory plus a rewriting engine to perform calculations and an automatic procedure for deciding (in)equalities. Our discussion will be general, but we will constantly refer, for the applicative aspects, to our particular implementation which uses Isabelle's higher-order logic theory (Isabelle/HOL) [10] and Standard ML [9].

Our tool is particularly designed to approach the formal verification of object code, since it was developed as a part of a bigger project, Holly [2], on this subject. Despite its origin, the designing purposes do not affect the generality of the ideas, nor the applicability of the implementation to other theories and problems.

The outline of the paper is as follows: Section 2 is devoted to the precise description of what computer arithmetic is; in Section 3, we specify each component of the system to deal with computer arithmetic; Section 4 explains how the combination of the previous parts allows us to tackle non trivial problems and how the interaction of components is the key for success in this application; in Section 5 we briefly discuss what we have achieved.

2 Computer Arithmetic

We said in the introduction that computer arithmetic is the logical theory which models the way computers treat numbers. In fact, we developed an abstract model for the arithmetic implemented by computers.

Every computer works by using bits, bytes, words and so on. These are the elements (interpreted as *numbers*) the Arithmetical Logical Unit (ALU, the part of a processor which is devoted to perform calculations [3]) is able to deal with. There are significant differences between processors in the *size* of numerical types (i.e., the precision) and in their *bounds* (i.e., the range of values an element can represent). Nowadays most processors are able to deal with operations on bytes, double bytes and quadruple bytes; the usual convention they use for the sign of a number is the two's complement representation, and they are able to perform additions, subtractions, multiplications, divisions, bitwise conjunction, bitwise disjunction and bitwise negation on the numerical types they provide.

We want to have a general theory, which closely resembles the way ALUs perform calculations, but we do not want to choose a particular set of numerical types for a specific processor. Since we aim to be very general, we adopt a simplifying decision: we specify only one numerical type, integers (formally represented as the type **int**), with no bounds and infinite precision, and we represent them in the two's complement notation, with the full set of standard operations. In section 3.1, we will provide the technical details.

We can define the computer numerical types by *subtyping* [1, 12]:

$$\begin{array}{ll}\textbf{unsigned\_bytes} & = \{x : \textbf{int} \mid 0 \leq x < 256\} \\ \textbf{signed\_bytes} & = \{x : \textbf{int} \mid -128 \leq x < 128\}\end{array} \quad ,$$

or by *quotienting* with an equivalence relation [10]:

$$\textbf{byte} = \textbf{int}/\{\langle x, y\rangle \mid x \bmod 256 = y \bmod 256\} \ .$$

If we do not consider the bitwise operations, **int** is isomorphic to $\mathbb{Z}$, the ring of integers numbers. In this way our system can be effectively used as an arithmetical oracle on integers.

We can even use our tool as an oracle for standard arithmetic, that is the theory of natural numbers: they could be represented by subtyping

$$\textbf{nat} = \{x : \textbf{int} \mid x \geq 0\} \ ,$$

or by quotienting

$$\textbf{nat} = \textbf{int}/\{\langle x, y\rangle \mid x = y \lor x = -y\} \ .$$

Both of these axiomatizations have some difficulties: the former does not respect subtraction, e.g., $3 - 7 = -4$ on integers, but $3 - 7$ should be 0 on naturals; the latter does not respect the ordering relation, e.g., $-3 < 2$ on integers, but $3 \not< 2$ on naturals.

For this reason, and because Isabelle provides a good implementation for the theory of natural numbers (terms of type *nat*), our tool is able to treat them directly.

Our system is developed as a package for the higher-order logic theory of the Isabelle theorem prover, so it seems worthwhile to compare it with similar theories/packages for Isabelle and other theorem provers.

Isabelle has no specialized decision procedures for the theory of integer numbers: its formalization of natural number arithmetic is quite advanced and a powerful simplification procedure is able to decide most elementary propositions in a reasonably efficient way. But, for the moment, the theory of integer numbers is quite undeveloped; for instance, it lacks the division and modulus operations, and, without modifying the simplifier, it fails to prove simple goals like

$$x + 1 - x = 1 \ .$$

The comparison with other, similar, theorem provers is more problematic, since their designs are different and it is not simple to identify a subsystem which could be properly compared with our tool. From a very general point of view, the HOL theorem prover [7] has a decision procedure (coded up as a tactic) which could be roughly equivalent to the calculation engine (Section 3.2), while the logical theory is an essential part of the logic the prover provides. There are no specialized decision procedures for arithmetic in HOL, but only clever instances of the tactics for the general logic.

The situation for PVS [8] is different: this theorem prover provides a decision procedure for integer arithmetic, which is based on the same algorithm we use (see Section 3.3); the logical representation for integers is part of the standard logic, and an equivalent of the calculational engine is embedded into the simplification tactics. But, in contrast with our tool, PVS does not provide bitwise operations on pure integers, but it develops a specialized theory (called *bitvector*) to reason about binary represented numbers.

3 The Components

The concept of computer arithmetic is implemented by three major parts. A *logical theory* for arithmetic is developed in section 3.1. It comprises a definition of integer numbers and operations on them using Isabelle/HOL [11] as an example environment. Section 3.2 presents the *calculation engine* used to do calculations and simplifications on arithmetical expressions, efficiently. Last but not least, a *decision procedure* is introduced in section 3.3. We have chosen SHOSTAK'S SUP-INF method [14] for proving formulas of an extension of quantifier-free Presburger arithmetic.

3.1 The Logical Development

Our first goal is to devise a logical theory which describes what we intend to call integer numbers, the definitions of operations and their basic properties. This theory is an instrument to reason, in a formal way, e.g., using a theorem prover, about properties of integers which could be modeled inside computer arithmetic.

As remarked in Section 2, computer arithmetic is based on a particular representation of numbers: every number is represented as a string of binary digits. Formally the type for integer numbers is defined as follows[1]:

$$\begin{aligned}\texttt{datatype}\ int = {} & \texttt{PlusSign} \\ \mid {} & \texttt{MinusSign} \\ \mid {} & \texttt{Bcons}\ int\ bool\end{aligned}$$

The value `PlusSign` stands for 0 while the value `MinusSign` stands for -1; looking at the binary representation, `PlusSign` is the infinite string where every bit is 0, while `MinusSign` is the infinite string where every bit is 1. The `Bcons` constructor works by appending a bit ($False$ for 0, and $True$ for 1) to a string of bits.

Intuitively, 6 (binary 110) will be represented as

$$\texttt{Bcons}\ (\texttt{Bcons}\ (\texttt{Bcons}\ \texttt{PlusSign}\ True)\ True)\ False$$

and -6 (binary $\ldots1010$) will be represented as

$$\texttt{Bcons}\ (\texttt{Bcons}\ (\texttt{Bcons}\ \texttt{MinusSign}\ False)\ True)\ False$$

The mapping from `int` to integers is straightforward:

$$\begin{aligned}&\texttt{PlusSign} && \stackrel{\text{map}}{\longmapsto} 0\\ &\texttt{MinusSign} && \stackrel{\text{map}}{\longmapsto} -1\\ &\texttt{Bcons}\ x\ y && \stackrel{\text{map}}{\longmapsto} 2(\text{map}\ x) + (\text{if}\ y\ \text{then}\ 1\ \text{else}\ 0)\quad ,\end{aligned}$$

and vice versa

$$\begin{aligned}&0 && \stackrel{\text{map}'}{\longmapsto} \texttt{PlusSign} && \\ &-1 && \stackrel{\text{map}'}{\longmapsto} \texttt{MinusSign} && \\ &2x && \stackrel{\text{map}'}{\longmapsto} \texttt{Bcons}\ (\text{map}'\ x)\ False && \text{if}\ x \neq 0\\ &2x+1 && \stackrel{\text{map}'}{\longmapsto} \texttt{Bcons}\ (\text{map}'\ x)\ True && \text{if}\ x \neq -1\quad .\end{aligned}$$

We will use, for clarity, the usual numerical representation[2], or a binary representation, where it is convenient. Both notations are unambiguous.

An important point about this representation lays in the fact that it is not fully determined: the same integer could be represented in different, but equivalent, ways. For example 5 could be represented by $\ldots0101$, or by $\ldots00000101$. The difference between two equivalent representation is in the trailing bits: formally,

$$\texttt{Bcons}\ \texttt{PlusSign}\ False = \texttt{PlusSign}$$

[1] This representation is borrowed from the `Bin` theory of Isabelle/HOL.

[2] So `PlusSign` becomes 0, `MinusSign` becomes -1, `Bcons` x $True$ is written as $2x+1$ and `Bcons` x $False$ is written as $2x$.

and

$$\texttt{Bcons MinusSign}\ True = \texttt{MinusSign} \ .$$

Propagating these equations through the inductive structure of the definition of *int*, we eventually get a normal form for our representation. In this way, an easy definition of equality is given: two representations are equal if and only if their normal forms are syntactically identical.

The definition of the successor and predecessor functions, as well as of addition, subtraction and multiplication is straightforward, and it is summarized in Figure 1.

Successor function

$$\begin{aligned}
\mathrm{Succ}(0) &= 1\\
\mathrm{Succ}(-1) &= 0\\
\mathrm{Succ}(2x+1) &= 2\ \mathrm{Succ}(x)\\
\mathrm{Succ}(2x) &= 2x+1
\end{aligned}$$

Predecessor function

$$\begin{aligned}
\mathrm{Pred}(0) &= -1\\
\mathrm{Pred}(-1) &= -2\\
\mathrm{Pred}(2x+1) &= 2x\\
\mathrm{Pred}(2x) &= 2\ \mathrm{Pred}(x)+1
\end{aligned}$$

Addition

$$\begin{aligned}
0+x &= x\\
-1+x &= \mathrm{Pred}(x)\\
x+0 &= x\\
x+-1 &= \mathrm{Pred}(x)\\
(2x+1)+(2y+1) &= 2\ \mathrm{Succ}(x+y)\\
(2x+1)+2y &= 2(x+y)+1\\
2x+(2y+1) &= 2(x+y)+1\\
2x+2y &= 2(x+y)
\end{aligned}$$

Numeric Complement

$$\begin{aligned}
-0 &= 0\\
-(-1) &= 1\\
-(2x+1) &= \mathrm{Pred}(-2x)\\
-2x &= 2\ (-x)
\end{aligned}$$

Subtraction

$$x-y = x+(-y)$$

Multiplication

$$\begin{aligned}
0\cdot x &= 0\\
-1\cdot x &= -x\\
(2x+1)\cdot y &= 2(x\cdot y)+y\\
2x\cdot y &= 2(x\cdot y)
\end{aligned}$$

Figure 1: Definitions for arithmetical operations

Formalizing division is more complex, since it is a partial operation, i.e., it is not defined when the divisor is 0. For the same reason, the remainder operation (mod) is awkward to define in a direct way. We define these pair of operators as the ones satisfying the basic equation:

$$x/y + x \bmod y = x \qquad \text{when } y \neq 0$$

under the additional constraint

$$0 \leq x \bmod y < y$$

Bitwise operations are easily defined by induction on the structure of integer representation: the precise statements are given in Figure 2.

Conjunction

$$
\begin{aligned}
0 \mathbin{\underline{\wedge}} x &= 0\\
-1 \mathbin{\underline{\wedge}} x &= x\\
x \mathbin{\underline{\wedge}} 0 &= 0\\
x \mathbin{\underline{\wedge}} -1 &= x\\
(2x+1) \mathbin{\underline{\wedge}} (2y+1) &= 2(x \mathbin{\underline{\wedge}} y)+1\\
(2x+1) \mathbin{\underline{\wedge}} 2y &= 2(x \mathbin{\underline{\wedge}} y)\\
2x \mathbin{\underline{\wedge}} (2y+1) &= 2(x \mathbin{\underline{\wedge}} y)\\
2x \mathbin{\underline{\wedge}} 2y &= 2(x \mathbin{\underline{\wedge}} y)
\end{aligned}
$$

Negation

$$
\begin{aligned}
\underline{\neg} 0 &= -1\\
\underline{\neg} -1 &= 0\\
\underline{\neg}(2x+1) &= 2(\underline{\neg} x)\\
\underline{\neg}(2x) &= 2(\underline{\neg} x)+1
\end{aligned}
$$

Disjunction

$$
\begin{aligned}
0 \mathbin{\underline{\vee}} x &= x\\
-1 \mathbin{\underline{\vee}} x &= -1\\
x \mathbin{\underline{\vee}} 0 &= x\\
x \mathbin{\underline{\vee}} -1 &= -1\\
(2x+1) \mathbin{\underline{\vee}} (2y+1) &= 2(x \mathbin{\underline{\vee}} y)+1\\
(2x+1) \mathbin{\underline{\vee}} 2y &= 2(x \mathbin{\underline{\vee}} y)+1\\
2x \mathbin{\underline{\vee}} (2y+1) &= 2(x \mathbin{\underline{\vee}} y)+1\\
2x \mathbin{\underline{\vee}} 2y &= 2(x \mathbin{\underline{\vee}} y)
\end{aligned}
$$

Figure 2: Definitions for bitwise operations

In order to reason about inequalities we need to define the ordering relation. This is done as a two-step process: first, we reduce the *less than* to 0 *is less than*

$$x < y \equiv \exists z. 0 < z \wedge x + z = y \quad ,$$

then, we state the conditions under which an integer is positive

$$
\begin{aligned}
0 &\not< 0\\
0 &\not< -1\\
0 < 2x &\equiv 0 < x\\
0 < 2x+1 &\equiv 0 < x \vee 0 = x \quad .
\end{aligned}
$$

In the usual way we define $x \leq y \equiv x < y \vee x = y$, $x \geq y \equiv y \leq x$ and $x > y \equiv y < x$.

In order to be able to deduce interesting properties about numbers, we need some induction principles: we provide the ones summarized in Figure 3. They encode induction over the structure of the binary representation, and two variants of the standard induction over natural numbers. These instruments are used to prove a wide set of lemmas which provides a (partial) validation for the other components, specifically for the rewrite rules of the calculation engine and the axioms of the decision procedure.

$$\frac{P(k) \qquad \begin{array}{c} P(x), x \leq k \\ \vdots \\ P(x+1) \end{array}}{\forall x \leq k.P(x)} \qquad \frac{P(k) \qquad \begin{array}{c} P(x), x \geq k \\ \vdots \\ P(x+1) \end{array}}{\forall x \geq k.P(x)}$$

$$\frac{P(0) \quad P(-1) \quad \begin{array}{c} P(x) \\ \vdots \\ P(2x) \end{array} \quad \begin{array}{c} P(x) \\ \vdots \\ P(2x+1) \end{array}}{\forall x.P(x)}$$

Figure 3: Induction principles

3.2 The Calculational Engine

A theory of integer numbers is developed in a logical framework; see section 3.1. The purpose of the Calculational Engine is to do reductions on integer expressions outside the logic.

A calculational engine works in a syntactical way by term rewriting and in a semantical way by doing actual calculations on integer numbers using the implementation programming language. It has been implemented in a functional programming language, namely *Standard ML* [9]. This engine is defined by a function that takes the representation of an integer expression and gives back a term that represents an equivalent, but reduced integer expression. We clarify next what we understand by *representation* of an integer expression, by *equivalent*, and by *reduced*.

The syntax of an internal language, in which the calculations and rewritings are done, is fixed by a data type. The corresponding construct `IntTerm` in ML is shown in Figure 4. It defines inductively a set of terms, called **Int** for the rest of this section.

Since the calculational engine is used to reason about integer expressions, that data type closely corresponds to the syntax for integer expressions used in section 3.1. In fact, to allow a smooth combination of deduction system and calculational engine, there is an obvious mapping from integer expressions to their representations in the internal language. That mapping is a bijection between the quotient of integer expressions formed by the

```
datatype IntTerm = IntConstant of term * IntTerm list
                 | IntValue of int
                 | IntSucc of IntTerm
                 | IntPred of IntTerm
                 | IntComp of IntTerm
                 | IntPlus of IntTerm * IntTerm
                 | IntMinus of IntTerm * IntTerm
                 | IntTimes of IntTerm * IntTerm
                 | IntDivide of IntTerm * IntTerm
                 | IntModulus of IntTerm * IntTerm;
```

Figure 4: The syntax of the internal language.

equivalence relation, that relates different representations of the same integer number, and the representation of integer expressions in ML. See section 4 for the combination of reasoners.

We call expressions, formed by `IntTerm`, integer expressions for the rest of this section, bearing in mind that those are just a *representation* of the expressions developed in 3.1.

The manipulation of integer expressions is done by a set of ML functions, called *rewrite functions*. Figure 5 shows such a function. The pattern matching feature of ML is used here to rewrite terms that have a certain structure and leave others unchanged.

```
fun Axiom1 (IntPlus (IntValue 0, x):  IntTerm) =
      x
  | Axiom1 default = default;
```

Figure 5: A rewrite function.

A *rewrite rule* is gained by taking a lemma about the equivalence of integer expressions deduced from the definition of integers on the logical level and directing the equation, say from left to right, then we also translate the left- and the right-hand side to **Int**, using the obvious mapping where integer variables are taken to ML variables to allow pattern matching. A rewrite function is made from such rules in the obvious way. The following shows the preparing steps to implement the rewrite function of figure 5. We take a lemma given by the definition of integers in section 3.1.

$$x + 0 = x$$

That translates to the rewrite rule

$$\texttt{IntPlus (}x\texttt{, IntValue 0)} \rightarrow x$$

where x is a ML variable over `IntTerm`.

This tight relation between rewrite functions in ML and lemmas derived on the logical level gives a good justification for the claim that an integer expression is rewritten into an expression which is equivalent by means of the logical definition of integers. A formal treatment of that matter is beyond the scope of this paper, textbooks on term rewriting or overview articles such as [5] may be referred to for the standard proofs necessary.

A crucial matter for the practical use of the calculational engine is the guarantee of termination. Since the rewrite procedure stops only if no more rules can be applied to any subterm of a given expression, it is therefore obvious that one cannot choose arbitrary lemmas to make them to rewrite rules. For instance, the unconditional application of

$$\texttt{IntPlus}\ (x,\ y) \rightarrow \texttt{IntPlus}\ (y,\ x)$$

is always possible if the argument expression contains addition.

The rewriting stops if every rewrite rule produces a "reduced" version of its argument, if applied. To express this reduction a well-founded ordering over **Int** has to be defined, and then one has to show that every rewrite rule in the system, if applied, gives a result that is strictly smaller than the argument. Provided that holds, the rewrite process must terminate, because the reduction cannot go on forever.

An ordering over terms can be conveniently defined by a homomorphism from the ground terms of **Int** to an algebra $\mathcal{A}$ (of the same signature) which already has a well founded ordering. Let $\succ$ be such a well founded ordering on $\mathcal{A}$, then the monotonicity condition:

$$f_{\mathcal{A}}(\ldots x \ldots) \succ f_{\mathcal{A}}(\ldots y \ldots) \quad \text{if} \quad x \succ y$$

for all operations $f_{\mathcal{A}}$ and all x, y in $\mathcal{A}$ has to hold, as well.

The mapping τ_1 : **Int** $\rightarrow \{2, 3, \ldots\}$ explained in Figure 6 is a good starting point for an overall termination argument, where $\succ$ is the usual well-ordering on natural numbers.

$$\begin{array}{lll}
\texttt{IntConstant}_{\mathcal{A}}(a,\ b) & = 2 & \texttt{IntPlus}_{\mathcal{A}}(a,\ b) & = a+b+1 \\
\texttt{IntValue}_{\mathcal{A}}\ a & = 2 & \texttt{IntMinus}_{\mathcal{A}}(a,\ b) & = a+2^b+1 \\
\texttt{IntSucc}_{\mathcal{A}}\ a & = a+4 & \texttt{IntTimes}_{\mathcal{A}}(a,\ b) & = a \cdot b \\
\texttt{IntPred}_{\mathcal{A}}\ a & = a+4 & \texttt{IntDivide}_{\mathcal{A}}(a,\ b) & = a \cdot b \\
\texttt{IntComp}_{\mathcal{A}}\ a & = 2^a & \texttt{IntModulus}_{\mathcal{A}}(a,\ b) & = a \cdot b
\end{array}$$

Figure 6: A reduction mapping.

A reduction ordering $\succ_{\tau_1}$ over **Int** is now given by:

$$s \succ_{\tau_1} t \quad \text{iff} \quad \tau_1(s) \succ \tau_1(t)$$

with s, $t \in$ **Int**. However, we need further reduction relations for coping with associative and commutative rewrite rules. Such measures are given by the mappings τ_2 and τ_3 sketched in Figure 7 and Figure 8, respectively, where a suitable ordering relation $<$ has to be defined.

$$\begin{array}{lcl} \texttt{IntPlus}_{\mathcal{A}}\ \texttt{(}a\texttt{, IntPlus}_{\mathcal{A}}\ \texttt{(}b\texttt{, }c\texttt{))} & = & 2 \\ \texttt{IntPlus}_{\mathcal{A}}\ \texttt{(IntPlus}_{\mathcal{A}}\ \texttt{(}a\texttt{, }b\texttt{), }c\texttt{)} & = & 3 \\ \ldots & & \end{array}$$

Figure 7: A reduction mapping.

$$\begin{array}{lclll} \texttt{IntPlus}_{\mathcal{A}}\ \texttt{(}a\texttt{, }b\texttt{)} & = & 2 & \text{if} & a < b \\ \texttt{IntPlus}_{\mathcal{A}}\ \texttt{(}a\texttt{, }b\texttt{)} & = & 3 & \text{if} & b < a \\ \ldots & & & & \end{array}$$

Figure 8: A reduction mapping.

The reduction ordering $\succ_{\mathcal{T}}$, which we use, is built from τ_1, τ_2, and τ_3 in the standard way of composing orderings by the lexicographical ordering of the cross-product. That is, given s and $t \in$ textbfInt, then $s \succ_{\mathcal{T}} t$ iff $s \succ_{\tau_1} t$, or in case $s = t$, then $s \succ_{\tau_2} t$, or in case $s = t$, then $s \succ_{\tau_3} t$. The termination of the calculational engine can be easily shown with $\succ_{\mathcal{T}}$.

The following example appeared when an actual correctness proof of a program was performed by the authors. It shows the effectiveness of reasoning with this rewrite engine in practice. We do not use the syntax of `IntTerm` here to increase readability. The expression:

$$c + ((3 + (f(x) + (-4))) + (-c))$$

is subsequently reduced to

$$\begin{array}{l} c + ((f(x) + (3 + (-4))) + (-c)), \\ c + ((f(x) + (-1)) + (-c)), \\ c + (f(x) + ((-1) + (-c))), \\ c + (f(x) + ((-c) + (-1))), \\ c + ((-c) + (f(x) + (-1))), \\ 0 + (f(x) + (-1)), \end{array}$$

and finally

$$f(x) + (-1).$$

The treatment of rewriting bitwise operations is done in the same way as the one of integer expressions. For reasons of space and readability, we do not go further into this here.

3.3 The Decision Procedure

The third part of our representation of Computer Arithmetic consists of a decision procedure for integer arithmetic with function symbols. We use SHOSTAK's SUP-INF algorithm [13, 14] for solving unquantified Presburger formulas which might also contain an unlimited number of function and predicate symbols.

Intuitively, Presburger formulas are those that can be built up from integers and variables over integers, addition, equality and inequality relations, and first-order logical connectives. The decision procedure we use here, see [14], operates on an extension of the quantifier-free version of Presburger arithmetic. This extension allows an unlimited number of n-ary function symbols $f^{(n)} : \mathbb{Z}^n \to \mathbb{Z}$, and n-ary predicate symbols $P^{(n)} : \mathbb{Z}^n$, with $0 \leq n$, in each formula. These symbols are treated as undefined, i.e., they are not interpreted. A small example of this kind of formula is the following:

$$x < f(y) \wedge f(y) \leq (x+1) \to (P(x) \leftrightarrow P(f(y) + (-1))) \ .$$

More generally, SHOSTAK's algorithm works on unquantified formulas of first-order logic which contain any function and predicate symbols over the set of integers. Those symbols are not interpreted except the function $+$ and the predicates $<$, $\geq$, $>$, and $=$. A more general treatment of decision procedures is given in [15] and [4].

The decision procedure is linked to the theory of integers in Isabelle/HOL by a straightforward translation from logical formulas to an internal language similarly to the way described in section 3.2. In addition to that, we allow multiplication with constants, for they can be rewritten as a finite sum, and also subtraction, since that can be rewritten as addition by complementing the second addend.

The decision procedure rejects terms which cannot be translated into the internal language, i.e., which are not recognized as formulas of the described extension of unquantified Presburger arithmetic. An exception is signaled in that case and the calling procedure, e.g., a proof tactic, has to cope with it.

If a formula was read in, the procedure gives a *true* or *false* as answer, depending on which conclusion it has reached.

4 The Combined System

The strength of our approach to computer arithmetic is the combination of systems that have different fields of application but which are chosen and implemented to work together in order to solve a demanding problem. Efficient interaction and a reasonable "division of labour" was a major design goal.

Stemming from the demand to reason about computer arithmetic, a theory was developed which fits these needs and overcomes limitations of existing solutions by extending them, i.e., providing the division and modulus operation and integrating bitwise operations on integers.

In order to make use of this theory we need a way to reason about it. The most basic way to do that is to use basic rewriting of goals by rules gained from the equational theory induced by the definition of integers. Isabelle provides a powerful rewrite engine, called *simplifier*, that supports the proving procedure in a sophisticated way. Nevertheless, its generality prevents the simplifier to be as efficient and useful as a specialized rewrite engine could be. Many arithmetical goals could only awkwardly or not at all be proved with that tool alone. Apart from rewriting, actual calculations like division cannot be done in a reasonable way in Isabelle. The calculational engine is used to remedy such problems.

Simplifying arithmetical expressions by calculations and simple rewriting is a basic way of reasoning but is too "primitive" to provide efficient tactics. SHOSTAK'S decision procedure is therefore used to solve more sophisticated goals when reasoning about Computer Arithmetic. It should be remarked that the Computational Engine is used to aid the decision procedure directly by rewriting an integer expression into a proper format such that the decision procedure can reason about it.

Technically, the integration of the calculational engine into Isabelle is done as a simple tactic that calls an *oracle* which maps an Isabelle term to the internal language of the calculational engine. Given the conclusion of the current subgoal, the oracle then provides the result of the calculations as a theorem which states that the argument and the reduced expression are equivalent. The subgoal and that theorem are then resolved.

Alternatively, the calculational engine is linked to the simplifier in Isabelle. By setting up a simplification procedure the external reasoner can aid the simplifier when rewriting complex subgoals which involve arithmetical expressions.

The decision procedure is linked to the theorem prover in a similar way as the calculational engine.

Finally, we can say that a logical theory of integers and bitwise operations provides a suitable basis for modeling Computer Arithmetic. The reasoning in that theory is made feasible by the tight coupling of a specialized rewrite engine, combined with calculation capabilities, and a powerful decision procedure.

5 Conclusions

A combined system for reasoning in computer arithmetic has been shown in this contribution. An integrated approach of logical, calculational and rewriting techniques makes it feasible to deduce complex theorems in computer arithmetic.

For the future, we plan to generalize the decision procedure in the way discussed in [15] and [4].

We would like to acknowledge the help of Dr. Sara Kalvala in improving the ideas and the form of this article.

A Examples

The following examples have been performed in Isabelle94-8 using version 1.0 beta 2 of CAT[3] (Computer Arithmetic Toolkit), developed by the authors.

The examples show the formulas that are reasoned about and the corresponding (slightly modified) dialog in the Isabelle/HOL environment.

Example 1

$P(z) \wedge z = 1 \wedge g(y) = z + 4 \rightarrow f(g(y)) = f(3 + 2z) \vee \neg P(1)$

```
> goal thy "((P(z) & z = #1) & g(y::int) = z + #4) --> \
\         ((f(g(y))::int) = f(#3 + #2 * z) | ~(P(#1)))";
Level 0
...
> by (supinf_tac 1);
Level 1
(P z & z = #1) & g y = z + #4 --> f (g y) = f (#3 + #2 * z) | ~ P #1
No subgoals!
```

Example 2

$\forall x.\forall y.(x < y - 4) \vee (x = y - 4) \vee (x = y - 3) \vee (x = y - 2) \leftrightarrow (x < y - 1)$

```
> goal ExtBin.thy "! x y. (x < y - #4 | \
\                         x = y - #4 | \
\                         x = y - #3 | \
\                         x = y - #2) =  (x < y - #1)";
Level 0
...
> by (supinf_tac 1);
Level 1
! x y. (x < y-#4 | x = y-#4 | x = y-#3 | x = y-#2) = (x < y-#1)
No subgoals!
```

The examples 3 and 4 are taken from [13].

[3] see http://www.dcs.warwick.ac.uk/holly/CAT

Example 3

The formula $\forall x.\forall y.2x + 3y \neq 1$ is not valid; proof level 0 and 1 are the same.

```
> goal ExtBin.thy "! x y. ~(#2 * x + #3 * y = #1)";
Level 0
! x y. #2 * x + #3 * y ~= #1
 1. ! x y. #2 * x + #3 * y ~= #1
...
> by (supinf_tac 1);
Warning: same as previous level
Level 1
! x y. #2 * x + #3 * y ~= #1
 1. ! x y. #2 * x + #3 * y ~= #1
```

Example 4

The formula $\forall x.\forall y.2x + 2y \neq 1$ is valid but SUP-INF cannot prove that. This example shows the incompleteness of the decision procedure.

```
> goal ExtBin.thy "! x y. ~(#2 * x + #2 * y = #1)";
Level 0
! x y. #2 * x + #2 * y ~= #1
 1. ! x y. #2 * x + #2 * y ~= #1
...
> by (supinf_tac 1);
Warning: same as previous level
Level 1
! x y. #2 * x + #2 * y ~= #1
 1. ! x y. #2 * x + #2 * y ~= #1
```

The following examples show the use of the calculational engine in CAT.

Example 5

The formula $(x + y)^2 = x^2 + 2xy + y^2$ is solved by reflexivity of "=" after it was rewritten.

```
> goal ExtBin.thy "(x + y) zexp #2 = x * x + #2 * x * y + y * y";
Level 0
...
> by (extbin_tac 1);
Level 1
(x + y) zexp #2 = x * x + #2 * x * y + y * y
No subgoals!
```

Example 6

The formula $x - 1 = y + (2 + (x + -3) + -y)$ is solved by reflexivity of "=" after it was rewritten.

```
> goal ExtBin.thy "x - #1 = y + (#2 + (x + $~ #3) + $~ y)";
Level 0
...
> by (extbin_tac 1);
Level 1
x - #1 = y + (#2 + (x + $~ #3) + $~ y)
No subgoals!
```

References

[1] P. Andrews. *An Introduction to Higher-Order Logic: to Truth through Proof.* Academic Press, New York, 1986.

[2] Marco Benini, Sara Kalvala, and Dirk Nowotka. Program abstraction in a higher-order logic framework. In *Proceedings of Theorem Proving in Higher-Order Logic '98 International Conference*. Theorem Proving in Higher-Order Logic '98, 1998. To appear.

[3] Edward L. Braun. *Digital Computer Design*. Academic Press, New York and London, 1963.

[4] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, number 1104 in Lecture Notes in Artificial Intelligence, pages 463–477, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[5] Nachum Dershowitz and Jean-Pierre Jounnaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 244–320. Elsevier Science Publishers, 1990.

[6] Michael J. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P. Subrahmanyan, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, Amsterdam, 1986. North Holland.

[7] Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[8] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[9] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[10] Lawrence C. Paulson. *Isabelle's Object-logics*, chapter Higher-Order Logic, pages 59–99. In [11], May 1997.

[11] Lawrence C. Paulson. Isabelle's object-logics. Technical Report 286, Computer Laboratory, University of Cambridge, May 1997.

[12] Lawrence C. Paulson. Isabelle's reference manual. Technical Report 283, Computer Laboratory, University of Cambridge, May 1997.

[13] Robert E. Shostak. On the SUP-INF method for proving presburger formulas. *JACM*, 24(4):529–543, October 1977.

[14] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *JACM*, 26(2):351–360, April 1979.

[15] Robert E. Shostak. Deciding combinations of theories. *JACM*, 31(1):1–12, January 1984.

[16] Craig Smorynski. *Logical Number Theory*, volume I. Springer-Verlag, 1991.