Reprinted from the *13th International Conference on Compiler Construction*

Widening Integer Arithmetic

Kevin Redwine and Norman Ramsey

Division of Engineering and Applied Sciences
Harvard University
{redwine,nr}@eecs.harvard.edu

Abstract. Some codes require computations to use fewer bits of precision than are normal for the target machine. For example, Java requires 32-bit arithmetic even on a 64-bit target. To run narrow codes on a wide target machine, we present a *widening* transformation. Almost every narrow operation can be widened by sign- or zero-extending the operands and using a target-machine instruction at its natural width. But extensions can sometimes be avoided, and our transformation avoids as many as possible. The key idea is knowing what each operation can accept in the high bits of its arguments and what it can guarantee about the high bits of its result. This knowledge is formalized using *fill types*, which drive the widening transformation.

1 Introduction

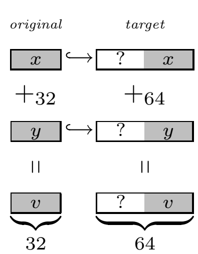

Fig. 1. Widening 32-bit addition: $x +_{32} y$

Machines can perform arithmetic and logical operations only on values of a particular width, often 32 or 64 bits. A compiler that knows the width of its target machine can generate code containing only operations of the correct width. But sometimes a different width is required by the source code. For example, Java requires 32-bit operations, but a compiler for IA-64 must generate 64-bit operations. Or if the source is binary code, as in a binary translator, then the width of operations is determined by the source machine. In general, if the source code requires widths not available on the target machine, a translator must change the widths without changing the meaning of the program. For example, a translated program might simulate 32-bit operations using 64-bit hardware. We call the relevant translation *widening*.

Take, for example, a program that adds two 32-bit values and produces a 32-bit result. A simple way to simulate 32-bit addition on a 64-bit machine is to widen the operand values to 64 bits and perform a 64-bit addition. The result of the original 32-bit addition is in the least significant, or low, 32 bits of the 64-bit result. This example is illustrated in Fig. 1. The heart of the widening problem is to determine how to fill in the high bits, which are shown as question marks in Fig. 1.

A simple solution is to explicitly sign extend (or zero extend) every variable and every intermediate result, ensuring that the wide and narrow representations have exactly the same denotation as a two's-complement (respectively unsigned) integer. But for many expressions, this would be overkill; for example, in the addition above, the correct value v appears in the low 32 bits of the result regardless of what is in the high bits of the operands. A good solution, therefore, minimizes the number of explicit sign extensions or zero extensions in the widened expression. This paper presents such a solution, which is based on the following contributions:

- To express the set of operations that can be implemented on a particular target machine, we use *bitwidth types* borrowed from λ-RTL (Ramsey and Davidson, 1998).
- To express mathematical truths about how different operations use and produce high bits, we present *fill types*. An operator has a fill type only if it is *widenable*, i.e., if applying the operator to wide values can simulate the same operator applied to narrow values. In this case, the operator's fill type formalizes what it requires in the high bits of its arguments and what it guarantees about the high bits of its result.
- To widen an expression while minimizing the number of explicit extensions, we present a translation expressed as a set of inference rules. The rules specify a translation of an input expression, which operates on narrow values, into an output expression, which operates on values that are supported by the target machine. Each rule is assigned a cost; a translation with minimal cost is computed by dynamic programming.

2 Machine-level computation at multiple widths

Our semantic model, which is based on the machine-description language λ-RTL (Ramsey and Davidson, 1998), is that a machine operates on bit vectors. It takes bit vectors from memory, registers, or immediate fields; it applies low-level operators to compute new bit vectors; and finally it stores the results in registers or memory. Bit vectors are untyped; type distinctions, such as the distinction between signed and unsigned integers, are entirely in the operators.

In λ-RTL, most operators are polymorphic in width. For example, xor accepts two n-bit values and produces an n-bit result, for any n. Polymorphism is essential for a metalanguage that must describe machines not yet built, but for widening, we need to reason only about particular instances of polymorphic operators: those that appear in the source code or that can be used on a particular target machine. Even one target machine may support multiple instances of an operator; for example, the Pentium provides three instances of integer addition, which operate on 8-bit, 16-bit, and 32-bit values.

As in λ-RTL, the type of an operator is determined by the width (number of bits) of its arguments and result. These widths can be used to classify operators.

– A typical operator takes two n-bit arguments and produces a result of the same width, or possibly a Boolean result, which in this paper we represent as a result of width 1.[1] We call such an operator an *ordinary value operator*.
– A less typical operator might produce take n-bit arguments and produce a result of a different width; for example, multiplying two n-bit arguments may produce a $2n$-bit result. Such an operator may change widths, but its primary purpose is to compute values, so we call it an *unusual value operator*.
– Three operators—sign extension, zero extension, and truncation—exist only to change widths. We call them *width-changing operators*.

The width-changing operators play a central role in widening. To make a narrow value wider, we can *sign extend* or *zero extend*.

sign extend $\mathrm{sx}_{w\leftarrow n}\, v$ fills the $(w-n)$ high bits of the result with copies of v's high (sign) bit. The subscript $_{w\leftarrow n}$ is pronounced "w from n."

$$v' = \mathrm{sx}_{8\leftarrow 4}\, v \qquad \begin{array}{r} \begin{array}{|c|c|c|c|}\hline v_3 & v_2 & v_1 & v_0 \\ \hline \end{array} = v \\ \begin{array}{|c|c|c|c|c|c|c|c|}\hline v_3 & v_3 & v_3 & v_3 & v_3 & v_2 & v_1 & v_0 \\ \hline \end{array} = v' \end{array}$$

zero extend $\mathrm{zx}_{w\leftarrow n}\, v$ fills the $(w-n)$ high bits of the result with zeroes.

$$v' = \mathrm{zx}_{8\leftarrow 4}\, v \qquad \begin{array}{r} \begin{array}{|c|c|c|c|}\hline v_3 & v_2 & v_1 & v_0 \\ \hline \end{array} = v \\ \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0 & v_3 & v_2 & v_1 & v_0 \\ \hline \end{array} = v' \end{array}$$

Each extension preserves the n least significant bits of its operand.

To make a wide value narrower, we can only *truncate*.

truncate $\mathrm{lo}_{n\leftarrow w}\, v$ extracts the n low bits of v. The high bits are lost.

$$v' = \mathrm{lo}_{4\leftarrow 8}\, v \qquad \begin{array}{r} \begin{array}{|c|c|c|c|c|c|c|c|}\hline v_7 & v_6 & v_5 & v_4 & v_3 & v_2 & v_1 & v_0 \\ \hline \end{array} = v \\ \begin{array}{|c|c|c|c|}\hline v_3 & v_2 & v_1 & v_0 \\ \hline \end{array} = v' \end{array}$$

Value operators at multiple widths If we are given a program that uses an operator at a narrow width n, but the target machine provides that operator only at a wider width w, what can we do? In almost all cases we can use the wide version of the operator. For example, 32-bit add can be implemented by 64-bit add, and 32-bit xor can be implemented by 64-bit xor, regardless of what is in the high bits of the operands. As another example, 32-bit unsigned divide can be implemented by 64-bit unsigned divide, but only if the high bits of the operands are zero. In general, different operators put different constraints on the high bits of their operands, so we have to know what is in those high bits. When operators

[1] Full λ-RTL includes a Boolean type, but in this paper we simplify by representing a Boolean value as a bit vector of width 1.

are nested, results become operands, so we also have to know about high bits of results. For example, if xor is given operands with zeroes in the high bits, it produces a result with zeroes in the high bits, and similarly for unsigned divide. But two's-complement addition produces a value whose high bits can't be relied on: In a typical case, the high bits may be sign bits, but if the add overflows, the high bits are garbage.

These properties of operators, operands, and results are truths of mathematics; they are independent of any program or machine. We express these truths using a type system. The type of a value tells us whether its high bits are filled with sign bits, zeroes, or garbage, so call the types *fill types*. The fill type of an operator tells us what it needs in the high bits of its operands and what it provides in the high bits of its result.

In translating an expression, we can force an operand to have the right fill type by inserting an explicit sign or zero extension. But if the operand already has the right fill type, we can do without an explicit extension. For example, the following two expressions produce the same result.

$$\begin{array}{l}\mathrm{divu}(\mathrm{zx}_{64\leftarrow 16}(\mathrm{lo}_{16\leftarrow 64}(\mathrm{divu}(\mathrm{zx}_{64\leftarrow 16}\,x_{16}, \mathrm{zx}_{64\leftarrow 16}\,y_{16}))), k_{64})\\ \mathrm{divu}(\qquad\qquad\qquad\qquad\quad \mathrm{divu}(\mathrm{zx}_{64\leftarrow 16}\,x_{16}, \mathrm{zx}_{64\leftarrow 16}\,y_{16})\quad, k_{64})\end{array}$$

The second expression avoids a superfluous zero extension.

Sometimes we can choose where to put an extension. Both xor and divu preserve zeroes in the high bits, so these expressions produce the same result.

$$\begin{array}{l}\mathrm{divu}(\mathrm{zx}_{64\leftarrow 32}(\mathrm{xor}(x_{32},\qquad\quad y_{32})),\qquad\quad k_{64})\\ \mathrm{divu}(\qquad\quad\ \mathrm{xor}(\mathrm{zx}_{64\leftarrow 32}\,x_{32}, \mathrm{zx}_{64\leftarrow 32}\,y_{32}), k_{64})\end{array}$$

Finally, many operators can be used at more than one fill type. If we choose fill types cleverly, we can avoid some extensions entirely. For example, in the somewhat fanciful expression

$$\mathrm{popcnt}(\mathrm{and}(\mathrm{neg}(x_{32}), \mathrm{divu}(y_{32}, k_{32}))),$$

popcnt counts the number of one bits in the value of a complex expression. To zero-extend each intermediate result would be valid but naïve. The constant k can be extended statically, so we need to extend at most variables x and y.

These examples show that different choices lead to different numbers of extensions in widened code. The number of extensions is the *cost* of widening, and we want to minimize it. Our implementation minimizes costs by dynamic programming (Sec. 7).

Which operators can be widened? Building on earlier work by Ramsey and Davidson (1998), we have identified 72 machine-level operators. These operators suffice to describe the instruction sets of a variety of machines, so chances are they include the operators one would find in any low-level intermediate code. Almost all operators are *widenable*, which means that a wide instance can do the job of a narrow instance (a formal definition appears in Sec. 3). There are 8 exceptions.

$$
\begin{array}{lrl}
\text{Expression} & e, b ::= & x_n \\
& \mid & k_n \\
& \mid & \mathrm{sx}_{w \leftarrow n}\, e \\
& \mid & \mathrm{zx}_{w \leftarrow n}\, e \\
& \mid & \mathrm{lo}_{n \leftarrow w}\, e \\
& \mid & \mathrm{sxlo}_{w \leftarrow n}(b, e) \\
& \mid & \mathrm{zxlo}_{w \leftarrow n}(b, e) \\
& \mid & \oplus_{n \leftarrow n_1 \times n_2}(e, e) \\
\text{Assignment } a & ::= & x_w :=_w e
\end{array}
$$

Metavariable	Stands for	Domain
e, b	expression	syntax
k	bit vector	syntax
v	bit vector	semantics
n, w	natural number	semantics
$\oplus$	binary operator	both

Metavariables n (narrow) and w (wide) give the widths of bit vectors. By convention, $n \leq n' \leq w \leq w'$.

Fig. 2. Abstract syntax and metavariable conventions of WL

– Bit-rotate left and bit-rotate right cannot be widened. A bit-rotate operator must be translated into a combination of shift left, shift right, and bitwise or operators, each of which can be widened individually.
– The 6 operators that detect overflow of integer addition, subtraction, multiplication, and division cannot be widened. Instead, they must be translated to detect "narrow overflow." Sec. 8 gives a sample translation.
– All 31 of the remaining integer-arithmetic, comparison, and logical operators can be widened, including the carry and borrow operators needed to implement multi-precision arithmetic.
– Of the 72 operators, 33 involve floating-point numbers. Floating-point operators can be widened (e.g., from 32 bits to 64 bits or from 64 bits to 80 bits), but the results produced by widened code only approximate the results from the original code, because intermediate results have more precision than originally specified. The details are beyond the scope of this paper.

Widening a program In practice, we widen not just a single operation but an entire program. A widened program uses only operators instantiated at widths that are actually supported by the hardware, but it has almost the same effect as the original program: In particular, the low bits of all variables are the same as in the original program. If the high bits of some variables are garbage, the widened program may leave different garbage in those high bits.

3 WL: A language to model widening

Because widening is a local transformation, widening a program is equivalent to widening every assignment in that program. We therefore model widening using a small language named WL, which has only assignments. An assignment has the form $x := e$, where the width of x is fixed by the program text, and operators in expression e may be nested arbitrarily deeply. Our translation handles assignments in which integer, logical, and comparison operators are used at any width up to the width of the target machine.

Fig. 2 shows the syntax of WL. An expression is a variable, bit-vector literal, sign or zero extension, truncation, or binary-arithmetic operation.

- A variable x_n stands for a mutable location that is n bits wide. On a typical machine, such a location is in a register or in memory.
- A literal k_n is a constant vector of n bits. We use decimal notation, but it stands for a two's-complement representation.
- A bit vector may be interpreted as a natural number (unsigned integer); we use the denotation function $\mathcal{U}_n$, where $\mathcal{U}_n(v) = \sum_{i=0}^{n-1} 2^i v_i$.
- Operation $\mathrm{sx}_{w \leftarrow n}\, e$ sign extends the n-bit expression e to w bits. Operation $\mathrm{zx}_{w \leftarrow n}\, e$ similarly zero extends e. Operation $\mathrm{lo}_{n \leftarrow w}\, e$ extracts the least significant n bits of the w-bit expression e. All three operations are explained in Sec. 2 above, with diagrams. On a typical machine, operations sx and zx are included in special-purpose load instructions such as "load signed byte" or "load halfword."

  In these width-changing operations, the left arrow is pronounced "from;" the right-to-left notation simplifies our proofs by making it easier to recognize and apply axioms such as $\mathrm{sx}_{w' \leftarrow w}(\mathrm{sx}_{w \leftarrow n}\, e) = \mathrm{sx}_{w' \leftarrow n}\, e$.
- Operations $\mathrm{sxlo}(b, e)$ and $\mathrm{zxlo}(b, e)$ extend the least significant b bits of expression e; to take an example, $\mathrm{sxlo}_{32 \leftarrow 32}(3_{32}, 7_{32}) = -1_{32}$. These operations can be thought of as composing extension with truncation, but they are not equivalent because b, the number of bits to be extended, may be computed dynamically. On a typical machine, the sxlo and zxlo operations do not change the widths of their operands, i.e., the only instances available are $\mathrm{sxlo}_{w \leftarrow w}$ and $\mathrm{zxlo}_{w \leftarrow w}$. Usually sxlo is implemented as left shift followed by arithmetic right shift, and zxlo is implemented as left shift followed by logical right shift.

  When the properties of the target machine are taken into account, sxlo may be more expressive than a combination of sx and lo. For example, the SPARC provides $\mathrm{sx}_{32 \leftarrow 8}$, $\mathrm{sx}_{32 \leftarrow 16}$, $\mathrm{lo}_{8 \leftarrow 32}$ and $\mathrm{lo}_{16 \leftarrow 32}$, and with these operators alone it is not possible to sign extend the least significant 13 bits of a 32-bit register. But the SPARC also provides $\mathrm{sxlo}_{32 \leftarrow 32}$, so we can perform this extension using $\mathrm{sxlo}_{32 \leftarrow 32}(13_{32}, e)$.
- To state properties that apply equally to both sx and zx or to sxlo and zxlo, we use the shorthands σx and σxlo.
- Operation $\oplus(e_1, e_2)$ applies the binary operator $\oplus$ to expressions e_1 and e_2. Operator $\oplus$ may stand for add, subtract, multiply, divide, xor, bitwise and, bitwise or, or other operators—a complete list appears in Fig. 4.

  We limit ourselves to binary operators only to keep notation simple. Our results apply to operators of any arity, and Fig. 4 presents our findings on all operators, not only binary ones.

WL's operational semantics appears in a companion technical report (Redwine and Ramsey, 2004).

4 Bitwidth types

Every expression in WL evaluates to a bit vector. On a particular target machine, bit vectors are restricted to certain widths. The width of the vector computed by an expression is given by that expression's *bitwidth type*. Our system of bitwidth types, which is a simplification of λ-RTL's type system (Ramsey and Davidson, 1998), involves two formal judgments. Judgment $M \vdash e : w$ says that in machine environment M, expression e is well typed with width w. Judgment $M \vdash x_w :=_w e$ says that assignment $x_w :=_w e$ is well typed in machine environment M. The rules are straightforward and are relegated to the companion technical report. The interesting part is the machine environment M.

The machine environment tells us at what widths each operator is available on a particular machine. Because an operator may be available at more than one type, we represent a machine environment M as a set of bindings of the form $\oplus : \mathit{width} \times \mathit{width} \to \mathit{width}$. For example, today's typical desktop machine might include the bindings $\mathrm{add} : 32 \times 32 \to 32$ and $\mathrm{lt} : 32 \times 32 \to 1$.

The machine environment M contains bindings only for instances available on a particular machine. We also define M^∞, the *complete* machine environment, which contains all possible instances of each polymorphic λ-RTL operator. M^∞ enables us to type-check an expression independent of the target machine; it is the price we pay for eliminating polymorphism from our formal system. For all target machine environments M, $M \subset M^\infty$.

If the bindings in a machine environment tell the truth about the operators, we call that environment *sound*.

Definition 1. M is *sound* iff whenever $\oplus : w_1 \times w_2 \to w \in M$, v_1 is w_1 bits wide, v_2 is w_2 bits wide, and $\oplus_{w \leftarrow w_1 \times w_2}(v_1, v_2) = v$, then value v is w bits wide.

An environment is *well formed* if for each operator, the widths of operands determine the width of the result. In a well-formed environment every expression has at most one bitwidth type.

Definition 2. M is *well formed* when, for any operator $\oplus$,

$$\oplus : w_1 \times w_2 \to w_3 \in M \land \oplus : w_1 \times w_2 \to w_4 \in M \text{ only if } w_3 = w_4 \,.$$

All environments in this paper, including M^∞, are sound and well formed.

5 Fill types for high bits

A fill type of an expression tells us about the high bits of the bit vector it evaluates to. If we know that the result of evaluating an expression will have sign bits, zeroes, or garbage in the high bits, we can give that expression a fill type of s, z, or g. Formally, to say that all but the least significant n bits of e are copies of the nth (sign) bit, we write $e :: \mathrm{s}[n]$. If all but the least significant n bits of e are zero, we write $e :: \mathrm{z}[n]$. If we know nothing about the high bits of

Fill-Var $(n \leq w)$
$$\frac{\Gamma(x_w) = \tau[n]}{\Gamma \vdash x_w :: \tau[n]}$$

Fill-Lit $(n \leq w,\ \mathrm{fits}_{\tau[n],w}\, k_w)$
$$\frac{}{\Gamma \vdash k_w :: \tau[n]}$$

Fill-Ext-Preserve $(n \leq n' \leq w)$
$$\frac{\Gamma \vdash e :: \sigma[n]}{\Gamma \vdash \sigma\mathrm{x}_{w \leftarrow n'}\, e :: \sigma[n]}$$

Fill-Ext $(n' \leq w)$
$$\frac{}{\Gamma \vdash \sigma\mathrm{x}_{w \leftarrow n'}\, e :: \sigma[n']}$$

Fill-Lo-Preserve $(n \leq n' \leq w)$
$$\frac{\Gamma \vdash e :: \tau[n]}{\Gamma \vdash \mathrm{lo}_{n' \leftarrow w}\, e :: \tau[n]}$$

Fill-Extlo-Preserve $(n \leq n' \leq w)$
$$\frac{\Gamma \vdash e :: \sigma[n]}{\Gamma \vdash \sigma\mathrm{xlo}_{w \leftarrow n'}(b, e) :: \sigma[n]}$$

Fill-Extlo $(n' \leq w)$
$$\frac{}{\Gamma \vdash \sigma\mathrm{xlo}_{w \leftarrow n'}(b, e) :: \sigma[n']}$$

Fill-Extlo-Lit
$$\frac{(n \leq n' \leq w,\ n = \mathcal{U}_{n'}(k_{n'}))}{\Gamma \vdash \sigma\mathrm{xlo}_{w \leftarrow n'}(k_{n'}, e) :: \sigma[n]}$$

Fill-Binop
$$\frac{\Gamma \vdash e_1 :: \tau_1[n_1] \qquad \Gamma \vdash e_2 :: \tau_2[n_2] \qquad \oplus :: \tau_1 \times \tau_2 \to \tau \qquad \oplus : n_1 \times n_2 \to n \in M^\infty}{\Gamma \vdash \oplus_{m \leftarrow m_1 \times m_2}(e_1, e_2) :: \tau[n]}$$

Fill-Garbage
$$\frac{}{\Gamma \vdash e :: \mathrm{g}[n]}$$

Fill-Subsume-Index $(n \leq n')$
$$\frac{\Gamma \vdash e :: \sigma[n]}{\Gamma \vdash e :: \sigma[n']}$$

Fig. 3. Fill typing rules

e's value, we can write $e :: \mathrm{g}[n]$ for any e. These types can be shown pictorially as follows:

$$\begin{array}{lll}
\text{High bits} & \text{Type} & \qquad\qquad n^{\text{th}}\text{ bit} \\
 & & \qquad\qquad\downarrow \\
\text{sign bits} & \mathrm{s}[n] & \boxed{b \dots b}\boxed{b}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot} \\
\text{zeroes} & \mathrm{z}[n] & \boxed{0 \dots 0}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot} \\
\text{anything} & \mathrm{g}[n] & \underbrace{\boxed{? \dots ?}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}}_{w}
\end{array}$$

Each fill type corresponds to a semantic predicate. Given a bit vector v of width w, $\mathrm{fits}_{\mathrm{s}[n],w}\, v$ is true iff the high $w - n$ bits of v are copies of the sign bit. Similarly, $\mathrm{fits}_{\mathrm{z}[n],w}\, v$ is true iff the high $w - n$ bits of v are zero. Formally, $\mathrm{fits}_{\mathrm{s}[n],w}\, v$ iff $v = \mathrm{sx}_{w \leftarrow n}(\mathrm{lo}_{n \leftarrow w}\, v)$ and $\mathrm{fits}_{\mathrm{z}[n],w}\, v$ iff $v = \mathrm{zx}_{w \leftarrow n}(\mathrm{lo}_{n \leftarrow w}\, v)$. Finally, $\mathrm{fits}_{\mathrm{g}[n],w}\, v$ is always true.

To abbreviate any fill type, we write τ instead of s, z, or g. To abbreviate only a non-garbage fill type, we write σ instead of s or z. The formal rules for fill types appear in Fig. 3. The formal judgment of the system is $\Gamma \vdash e :: \tau[n]$, where Γ is an environment that gives the fill type of each variable.

– A variable has the fill type assigned to it by Γ. We explain where Γ comes from in Sec. 6.1.

- A constant may have any fill type consistent with its value.
- An extension may preserve an existing fill type or establish a new fill type.
- Truncation preserves fill type, provided not too many bits are truncated.
- Just like $\sigma\mathrm{x}_{w\leftarrow n'}$, $\sigma\mathrm{xlo}_{w\leftarrow n'}$ may preserve an existing fill type or establish a new one. If the first argument is a literal constant, $\sigma\mathrm{xlo}_{w\leftarrow n'}$ may also establish a stronger fill type.
- The rule for binary operators is the most complex and also the most important rule; it is discussed below.
- The rules below the black bar express subtyping relations on fill types. The subtyping relations are a consequence of the fits predicate defined above. Garbage subsumes other types since it is always true. The meaning of the extension operations implies that if $\mathrm{fits}_{\sigma[n],w}$ then $\mathrm{fits}_{\sigma[n+1],w}$ for $n < w$, which explains the index subtypes.

Fill types of operators If a binary operator has a fill type, it takes the form $\tau_1 \times \tau_2 \to \tau$. The fill type is meaningful in conjunction with the operator's bitwidth type: If the operator is instantiated at bitwidth type $n_1 \times n_2 \to n$, then provided the arguments have fill types $\tau_1[n_1]$ and $\tau_2[n_2]$, the result has fill type $\tau[n]$. We give an operator a fill type only if that operator is *widenable*. Widenability promises not only that the result has the right fill type, but also that the least significant n bits of the result are the bits that would have been produced by applying the narrow instance of the operator to the least significant bits of the operands.

Definition 3. An operator $\oplus$ is *widenable* at fill type $\tau_1 \times \tau_2 \to \tau$, written $\oplus :: \tau_1 \times \tau_2 \to \tau$, if whenever M^∞ contains the two bindings $\oplus : n_1 \times n_2 \to n$ and $\oplus : w_1 \times w_2 \to w$, where $n_1 \leq w_1$, $n_2 \leq w_2$, and $n \leq w$, then for all v_1, v_2 such that $\mathrm{fits}_{\tau_1[n_1],w_1}\, v_1$ and $\mathrm{fits}_{\tau_2[n_2],w_2}\, v_2$, both $\mathrm{fits}_{\tau[n],w}(\oplus_{w\leftarrow w_1\times w_2}(v_1, v_2))$ and

$$\mathrm{lo}_{n\leftarrow w}(\oplus_{w\leftarrow w_1\times w_2}(v_1, v_2)) = \oplus_{n\leftarrow n_1\times n_2}(\mathrm{lo}_{n_1\leftarrow w_1}\, v_1, \mathrm{lo}_{n_2\leftarrow w_2}\, v_2)\,.$$

The second condition should also be taken to mean that $\oplus_{w\leftarrow w_1\times w_2}(v_1, v_2)$ is well defined whenever $\oplus_{n\leftarrow n_1\times n_2}(\mathrm{lo}_{n_1\leftarrow w_1}\, v_1, \mathrm{lo}_{n_2\leftarrow w_2}\, v_2)$ is: If a narrow computation completes successfully, so does a wide one.

Widenability is needed to show that the rules in Fig. 3 correctly predict the high bits of results—a property we call fill-type soundness. Soundness also requires a dynamic environment that is consistent with the fill-type environment: each variable must have the fill type claimed. Formally, ρ is *consistent* with Γ iff $\forall x_w \in \mathrm{dom}\,\Gamma, x_w \in \mathrm{dom}\,\rho \wedge \mathrm{fits}_{\Gamma(x_w),w}(\rho(x_w))$.

Theorem 4. (Fill-Type Soundness) *If $\Gamma \vdash e :: \tau[n]$, $\rho \vdash e \Downarrow v$, $M^\infty \vdash e : w$, and ρ is consistent with Γ, then* $\mathrm{fits}_{\tau[n],w}\, v$.

Widenability is a key idea: If an operator is widenable at a suitable fill type, a wide instance of the operator can do the job of a narrow instance. The examples in Sec. 2 are possible only because operators popcnt, neg, divu, bitwise and, and xor are widenable. The fill types of a widenable operator express all

add ::	$g \times g \rightarrow g$
add_overflows	Not widenable
and ::	$s \times s \rightarrow s$
and ::	$z \times g \rightarrow z$
and ::	$g \times z \rightarrow z$
and ::	$g \times g \rightarrow g$
borrow ::	$s \times s \times g \rightarrow z$
borrow ::	$z \times z \times g \rightarrow z$
carry ::	$s \times s \times g \rightarrow z$
com ::	$s \rightarrow s$
com ::	$g \rightarrow g$
div ::	$s \times s \rightarrow s$
div_overflows	Not widenable
divu ::	$z \times z \rightarrow z$
eq ::	$s \times s \rightarrow z$
eq ::	$z \times z \rightarrow z$
ge ::	$s \times s \rightarrow z$
geu ::	$s \times s \rightarrow z$
geu ::	$z \times z \rightarrow z$
gt ::	$s \times s \rightarrow z$
gtu ::	$s \times s \rightarrow z$
gtu ::	$z \times z \rightarrow z$
le ::	$s \times s \rightarrow z$
leu ::	$s \times s \rightarrow z$
leu ::	$z \times z \rightarrow z$
lt ::	$s \times s \rightarrow z$
ltu ::	$s \times s \rightarrow z$
ltu ::	$z \times z \rightarrow z$
mod ::	$s \times s \rightarrow s$
modu ::	$z \times z \rightarrow z$
mul ::	$s \times s \rightarrow s$
mulu ::	$z \times z \rightarrow z$
mul_overflows	Not widenable
mulu_overflows	Not widenable
ne ::	$s \times s \rightarrow z$
ne ::	$z \times z \rightarrow z$
neg ::	$g \rightarrow g$
or ::	$s \times s \rightarrow s$
or ::	$z \times z \rightarrow z$
or ::	$g \times g \rightarrow g$
popcnt ::	$z \rightarrow z$
quot ::	$s \times s \rightarrow s$
quot_overflows	Not widenable
rem ::	$s \times s \rightarrow s$
rotl	Not widenable
rotr	Not widenable
shl ::	$g \times z \rightarrow g$
shra ::	$s \times z \rightarrow s$
shrl ::	$z \times z \rightarrow z$
sub ::	$g \times g \rightarrow g$
sub_overflows	Not widenable
xor ::	$s \times s \rightarrow s$
xor ::	$z \times z \rightarrow z$
xor ::	$g \times g \rightarrow g$

Fig. 4. Widenability and fill types of integer and logical operators

the mathematical facts we need to know to widen expressions involving that operator. Fig. 4 lists the fill types and widenability of all non-floating-point operators. The fill type used for the results of comparisons is arbitrary, because a 1-bit value is always at its natural width. We have chosen the z type.

A few operators can illustrate the interpretation of fill types. Unsigned division (divu) has a fill type typical of unsigned operators. It requires both arguments to have zeroes in the high bits and guarantees a result with zeroes in the high bits. Addition (add) requires nothing of the high bits of its operands but guarantees nothing about the high bits of its result. Exclusive or (xor) is overloaded. It can accept any input, but guarantees to produce zeroes in the high bits of its result only if the high bits of each operand are zero and sign bits only if each operand has sign bits. Unsigned greater-than (gtu), perhaps surprisingly, is also overloaded: it can be used to compare sign-extended numbers as well as zero-extended numbers. It is an accident of two's-complement representation that extending the sign bit does not change the results of this comparison.

6 Translating a narrow program for a wide machine

To translate a narrow program to run on a wide machine takes two steps:

Variable placement: Decide in what location on the target machine each source variable will be stored.

Widening: Rewrite the program to use wide operators in place of narrow operators.

Although variable placement and widening can be combined, it is simpler to place variables before widening, as we have done in our implementation.

6.1 Placing variables

A variable placer chooses a location for each variable in the source program. Given a variable x_n of width n, there are two choices: place it in a target location of width n, which we also call x_n, or place it in the low n bits of a target location of width w, which we call x_w. (A location of width w may be used only if the variable stands for a location in a machine register; a variable placer should not change the layout of data structures in memory.) If we choose the low n bits of x_w, we have to decide what to do with the high bits. We express this decision by giving x_w a fill type, which we store in type environment Γ. For example, a Java signed 32-bit integer variable x might be placed in a 64-bit location, in which case $\Gamma(x_{64}) = \mathrm{s}[32]$.

The best choice of $\Gamma(x_w)$ depends on how x_w is used in the program. For example, if x_w is frequently passed to operators that require sign bits in the high bits, it may be best if $\Gamma(x_w) = \mathrm{s}[n]$. If x_w is frequently passed to operators that place no requirements on high bits, $\Gamma(x_w) = \mathrm{g}[n]$ may be best. Finding the best placement requires global knowledge about how each variable is used in each assignment and how frequently each assignment is executed.

Another interesting way to choose $\Gamma(x_w)$ is to use bitwidth inference (Razdan and Smith, 1994; Stephenson et al., 2000; Budiu et al., 2000). Bitwidth inference finds the range of values each variable can have. If the range is nonnegative, we use the z type; otherwise we use the s type. In either case, the range is used to determine the minimum bitwidth necessary to represent the variable. This bitwidth becomes the index of the variable's fill type $\Gamma(x_w)$.

Kawahito et al. (2002) describe a clever variation on placement that can be used to improve performance. At a definition of x_w, they assume $\Gamma(x_w) = \mathrm{s}[n]$, but at a use, they assume $\Gamma(x_w) = \mathrm{g}[n]$. These assumptions can yield a program with extension operations at both definitions and uses, which would be redundant. The advantage is that the redundancy can be removed from *either* the uses or the definition, whichever is more frequently executed. Our method can support this technique, but to keep things clear we suppress the details.

6.2 Widening

After a location has been chosen for each variable, the *widener* replaces each narrow operation with a wide operation or combination of operators. An operator that is widenable can simply be used at the new width, provided its operands are given suitable fill types. An operator that is not widenable must be rewritten as a combination of widenable operators. As discussed in Sec. 2, there are only a handful of non-widenable operators, and each can easily be rewritten.

Widening is defined by a formal system that yields judgments of the form

$$\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \tau[n] : w.$$

This judgment says that given variable placement Γ and target machine M, expression e can be widened to expression e' at cost c, and that e' has fill type $\tau[n]$ and width w. A similar judgment, $\Gamma, M \vdash a \underset{c}{\hookrightarrow} a'$, applies to assignments.

Given Γ, M, e, τ, n, and w, the rules in Fig. 5 find every e' that can be found by inserting sign or zero extensions. Our implementation uses dynamic programming to find an e' of minimum cost.

The rules in Fig. 5 are divided into four groups.

– The upper-left rules translate those parts of the abstract syntax that relate to value computation without widening or narrowing. The assignment rule, in particular, shows that the left-hand side's width w and fill type $\tau[n]$ provide the information needed to translate the right-hand side. The LIT rule shows that we can extend a literal constant at translation time. The BINOP rule is the primary source of nondeterminism in the system. Nondeterminism arises for those operators that have multiple fill types, such as exclusive or. Nondeterminism may also arise for an operator that the target machine supports at multiple widths. For example, if the source code contains a 12-bit add and the target machine is the Pentium, the widener can do the add at either 16 or 32 bits.

$\textsc{Assign}\ (n \leq w)$
$$\frac{\begin{array}{c}\Gamma(x_w) = \tau[n] \\ \Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \tau[n] : w\end{array}}{\Gamma, M \vdash x_w :=_w e \underset{c}{\hookrightarrow} x_w :=_w e'}$$

$\textsc{Var}\ (n \leq w)$
$$\frac{\Gamma(x_w) = \tau[n]}{\Gamma, M \vdash x_w \underset{0}{\hookrightarrow} x_w :: \tau[n] : w}$$

$\textsc{Lit}\ (k'_w = \sigma\mathrm{x}_{w \leftarrow n}\, k_n, n \leq w)$
$$\frac{}{\Gamma, M \vdash k_n \underset{0}{\hookrightarrow} k'_w :: \sigma[n] : w}$$

$\textsc{Binop}\ (n_1 \leq w_1, n_2 \leq w_2, n \leq w)$
$$\frac{\begin{array}{c}\oplus :: \tau_1 \times \tau_2 \rightarrow \tau \\ \oplus : n_1 \times n_2 \rightarrow n \in M^{\infty} \\ \oplus : w_1 \times w_2 \rightarrow w \in M \\ \Gamma, M \vdash e_1 \underset{c_1}{\hookrightarrow} e'_1 :: \tau_1[n_1] : w_1 \\ \Gamma, M \vdash e_2 \underset{c_2}{\hookrightarrow} e'_2 :: \tau_2[n_2] : w_2\end{array}}{\begin{array}{r}\Gamma, M \vdash \oplus_{n \leftarrow n_1 \times n_2}(e_1, e_2) \underset{c_1+c_2}{\hookrightarrow} \\ \oplus_{w \leftarrow w_1 \times w_2}(e'_1, e'_2) :: \tau[n] : w\end{array}}$$

$\textsc{Fill}\ (\mathcal{U}(k_w) = n, n \leq w \leq w')$
$$\frac{\begin{array}{c}\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \mathrm{g}[n] : w \\ \sigma\mathrm{xlo}_{w' \leftarrow w} \in M\end{array}}{\Gamma, M \vdash e \underset{c+1}{\hookrightarrow} \sigma\mathrm{xlo}_{w' \leftarrow w}(k_w, e') :: \sigma[n] : w'}$$

$\textsc{Widen-}\sigma\ (n \leq w \leq w')$
$$\frac{\begin{array}{c}\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \sigma[n] : w \\ \sigma\mathrm{x}_{w' \leftarrow w} \in M\end{array}}{\Gamma, M \vdash e \underset{c+1}{\hookrightarrow} \sigma\mathrm{x}_{w' \leftarrow w}\, e' :: \sigma[n] : w'}$$

$\textsc{Widen-g}\ (n \leq w \leq w')$
$$\frac{\begin{array}{c}\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \mathrm{g}[n] : w \\ \sigma\mathrm{x}_{w' \leftarrow w} \in M\end{array}}{\Gamma, M \vdash e \underset{c+1}{\hookrightarrow} \sigma\mathrm{x}_{w' \leftarrow w}\, e' :: \mathrm{g}[n] : w'}$$

$\textsc{Narrow}\ (n \leq w \leq w')$
$$\frac{\begin{array}{c}\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \tau[n] : w' \\ \mathrm{lo}_{w \leftarrow w'} \in M\end{array}}{\Gamma, M \vdash e \underset{c+1}{\hookrightarrow} \mathrm{lo}_{w \leftarrow w'}\, e' :: \tau[n] : w}$$

$\textsc{Drop-Ext}\ (n \leq n' \leq w)$
$$\frac{\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \sigma[n] : w}{\Gamma, M \vdash \sigma\mathrm{x}_{n' \leftarrow n}\, e \underset{c}{\hookrightarrow} e' :: \sigma[n] : w}$$

$\textsc{Drop-Lo-Copy}\ (n \leq n' \leq w)$
$$\frac{\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \tau[n'] : w}{\Gamma, M \vdash \mathrm{lo}_{n \leftarrow n'}\, e \underset{c}{\hookrightarrow} e' :: \tau[n'] : w}$$

$\textsc{Drop-Lo-Ignore}\ (n \leq n' \leq w)$
$$\frac{\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \tau[n'] : w}{\Gamma, M \vdash \mathrm{lo}_{n \leftarrow n'}\, e \underset{c}{\hookrightarrow} e' :: \mathrm{g}[n] : w}$$

$\textsc{Extlo}\ (n \leq n' \leq w \leq w')$
$$\frac{\begin{array}{c}\sigma\mathrm{xlo}_{w' \leftarrow w} \in M \\ \Gamma, M \vdash b \underset{c_1}{\hookrightarrow} b' :: \mathrm{z}[n] : w \\ \Gamma, M \vdash e \underset{c_2}{\hookrightarrow} e' :: \mathrm{g}[n] : w\end{array}}{\begin{array}{r}\Gamma, M \vdash \sigma\mathrm{xlo}_{n' \leftarrow n}(b, e) \underset{c_1+c_2+1}{\hookrightarrow} \\ \sigma\mathrm{xlo}_{w' \leftarrow w}(b', e') :: \sigma[n] : w'\end{array}}$$

$\textsc{Subsume-Fill}\ (n \leq w)$
$$\frac{\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \sigma[n] : w}{\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \mathrm{g}[n] : w}$$

$\textsc{Subsume-Index}\ (n < n' \leq w)$
$$\frac{\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \sigma[n] : w}{\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \sigma[n'] : w}$$

$\textsc{Natural}$
$$\frac{\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \mathrm{g}[w] : w}{\Gamma, M \vdash e \underset{c}{\hookrightarrow} e' :: \sigma[w] : w}$$

Fig. 5. Widening rules

The Binop rule shows where our two type systems come together. The fill types of an operator are universal truths of mathematics, but the bitwidth types of that operator are accidental facts about the machine.
The Binop rule combines both kinds of information.

- The lower-left rules provide the means of changing an expression's width. Each such rule requires an explicit extension or truncation. We need both Fill and Widen-σ because, as noted in Sec. 3, we may have to use either σxlo or σx depending on what is available on the target machine. Each of these rules adds 1 to the cost of widening because it inserts an extension or truncation operation.
- The upper-right rules cope with extensions and truncations in the source program. If the number of bits extended or truncated is known statically, we simply drop the extension or truncation and let the other rules insert one if needed. In rule Extlo, the number of bits extended is computed dynamically, so the extension must be retained. The Drop rules maintain the same cost because they insert nothing in the widened expression. Extlo, on the other hand, has to add the extension from the source expression into the widened expression.
- The lower-right rules permit us to exploit the subtyping rules for fill types. They change the type without changing the widened expression, so they have no extra cost.

The rules are sound, which means that a translated expression e' produces a value with the fill type claimed and with the right answer in the low bits. Also, all the operators in any translated expression are available on the target machine. Stating these properties precisely requires a bit more formalism.

To formalize soundness of the rules for assignment, we must also account for potential differences in high bits when $\Gamma(x_w) = \mathrm{g}[n]$.

Definition 5. Environments ρ and ρ' *agree up to widths in* Γ, written $\rho \sim_\Gamma \rho'$, when both ρ and ρ' are consistent with Γ and when for each variable x_w, $\rho(x_w)$ and $\rho'(x_w)$ are equal in their n low bits, where n is determined by $\Gamma(x_w)$. Formally, $\forall x_w \in \mathrm{dom}\,\Gamma, \mathrm{lo}_{n\leftarrow w}(\rho(x_w)) = \mathrm{lo}_{n\leftarrow w}(\rho'(x_w))$, where $\Gamma(x_w) = \tau[n]$.

N.B. If $\Gamma(x_w) = \mathrm{s}[n]$, then the high bits are equal to the nth bit, and because ρ and ρ' agree on the low n bits, $\rho(x_w)$ and $\rho'(x_w)$ must be identical. Similarly, if $\Gamma(x_w) = \mathrm{z}[n]$, then the high bits are zero, and again $\rho(x_w)$ and $\rho'(x_w)$ must be identical. But if $\Gamma(x_w) = \mathrm{g}[n]$, the high bits can differ before and after widening.

Soundness of translation says that a translated assignment produces almost the same results as the original, the only differences being those permitted to two environments that agree up to widths in Γ. The assignment soundness theorem is shown here pictorially as a commutative diagram.

$$\begin{tikzcd} \rho_{pre} \arrow[r, phantom, "\sim_\Gamma"] \arrow[d, Rightarrow, "x_w:=e"'] & \rho'_{pre} \arrow[d, Rightarrow, "x_w:=e'"] \\ \rho_{post} \arrow[r, phantom, "\sim_\Gamma"] & \rho'_{post} \end{tikzcd}$$

To prove soundness, we have a lemma for expressions. Whenever the original expression has a value, the translated expression does too, and the value of the translated expression agrees with the original on the low bits and respects the semantic predicate required by its fill type.

Theorem 6. (Assignment Soundness) *If $\Gamma, M \vdash x_w :=_w e \hookrightarrow x_w :=_w e'$, and we are given any ρ_1 and ρ_2 such that $\rho_1 \sim_\Gamma \rho_2$, and if $\rho_1 \vdash x_w :=_w e \Downarrow \rho_1'$, then $\rho_2 \vdash x_w :=_w e' \Downarrow \rho_2'$ and $\rho_1' \sim_\Gamma \rho_2'$.*

Lemma 7. (Expression Soundness) *For any sound and well-formed M and ρ consistent with Γ, if $\Gamma, M \vdash e \hookrightarrow e' :: \tau[n] : w$, $\rho \vdash e \Downarrow v$, and $M^\infty \vdash e : m$, then $\rho \vdash e' \Downarrow v'$, $\mathrm{fits}_{\tau[n],w}(v')$, and $n \leq m$ implies $\mathrm{lo}_{n \leftarrow w}\, v' = \mathrm{lo}_{n \leftarrow m}\, v$.*

Proof is by induction on the structure of derivations.

Soundness relies on the claims made about widenability of operators (Fig. 4). We make these claims based on careful thought, confirmed by exhaustive machine checking on a subset of the operators at widths 8 and 16. A formal proof would be better, but such a proof would require a bit-level semantics for each operator, such as is provided by Bacon (2001), as well as substantial theorem-proving expertise. This sort of treatment is well beyond the scope of this paper or of a typical compiler; in our opinion, the game is not worth the candle.

The translation is not only sound but also has other useful properties: e' has the types claimed for it, and e' can be implemented on the target machine.

7 Implementation and experimental results

We have implemented widening by using dynamic programming to compute the minimum-cost translation for a given expression at a given width and fill type. Dynamic programming is certain to find a solution if one exists, always finds the best solution, and is easy to implement by following the structure of our rules. The critical implementation decisions were how to represent a solution and how to eliminate redundant solutions. These details are covered in the companion technical report (Redwine and Ramsey, 2004).

We compare the dynamic-programming implementation with a top-down, greedy algorithm. The greedy algorithm inserts sign extensions or zero extensions below each operator as required by that operator's fill type. If an operator has more than one fill type, the greedy algorithm picks one that requires fewer extensions immediately below the operator. The greedy algorithm may seem simpler than dynamic programming, but in practice, it was surprisingly difficult to write and to think about, and we are less confident in its correctness.

We added both algorithms to the Quick C-- compiler, a compiler for C-- (Peyton Jones et al., 1999; Ramsey and Peyton Jones, 2000). The implementation is about 600 lines of Objective Caml code, including both algorithms and some supporting infrastructure. We then widened some 32-bit programs from a small test suite distributed with the LCC compiler (Fraser and Hanson, 1995). For this preliminary experiment, we placed variables in memory with garbage in

File name	Original	Greedy	DP	Reduction
incr.c	17	51	30	24%
struct.c	20	70	36	49%
front.c	31	93	58	38%
8q.c	36	113	66	42%
sort.c	61	187	117	37%
wf1.c	65	201	124	38%
init.c	77	239	148	38%
cq.c	79	252	151	40%
stdarg.c	112	392	215	45%
yacc.c	118	350	209	40%
switch.c	120	369	235	36%
limits.c	213	681	422	38%

Table 1. Dynamic programming versus a greedy algorithm on a subset of the LCC compiler test suite. Original column counts operations in WL produced by the compiler. Greedy and DP show the number of operations after widening (smaller is better). Reduction shows the percent reduction in operations, going from Greedy to DP.

the high bits, so for any x, $\Gamma(x) = \mathrm{g}[32]$. Quick C-- generates such poor code that measurements of run time are not meaningful, so as a proxy we evaluate effectiveness by counting the number of operations in the widened code.

Table 1 shows the results. The "Original" column shows the number of operations in original code. The "Greedy" and "DP" columns show the total number of operations in widened code. The final column shows the percent reduction in the number of operations going from the greedy algorithm to the dynamic-programming algorithm. The reductions range from 24% to 49%, showing that the dynamic-programming algorithm produces much better results.

8 Discussion

We have identified several potential extensions to our system, but none of significant value. For example, it is possible to define a fill type $o[n]$, which means that a value has ones in all but the least significant n bits. While the Boolean operators preserve the o type in ways one might expect (dual to the z type), it is not useful because no operation requires an operand with ones in the high bits.

Another potential extension is to apply widening to floating-point computations. Sign extension and truncation have their analogs in instructions for floating-point conversions, but there is a big difference: If a floating-point computation is performed with extra precision and truncated only at the end, we get a different answer than if every intermediate result is truncated to use less precision. If exact answers are required, then, one cannot remove any explicit extensions or truncations, and our ideas offer no benefit relative to a naïve translation. If, however, the programmer wants approximate answers fast, one can perform entire computations at higher precision. Our C-- back end for the Pentium offers this option, widening 32-bit and 64-bit floating-point computations to the full 80 bits offered by the hardware.

Even without these extensions, it is surprising how much machinery is needed to solve a seemingly simple problem. We started out with much simpler ideas, among which the central one was to rewrite expressions by exploiting algebraic laws such as the following:

$$
\begin{aligned}
\mathrm{lo}_{n\leftarrow w}(\mathrm{sx}_{w\leftarrow n}\, x) &= x \\
\mathrm{lo}_{n\leftarrow w}(\mathrm{zx}_{w\leftarrow n}\, x) &= x \\
\mathrm{sx}_{w\leftarrow n}(\mathrm{mul}(x,y)) &= \mathrm{mul}(\mathrm{sx}_{w\leftarrow n}\, x, \mathrm{sx}_{w\leftarrow n}\, y) \\
\mathrm{zx}_{w\leftarrow n}(\mathrm{mulu}(x,y)) &= \mathrm{mulu}(\mathrm{zx}_{w\leftarrow n}\, x, \mathrm{zx}_{w\leftarrow n}\, y)
\end{aligned}
$$

Unfortunately, simple algebraic rewriting has two deficiencies:

- It is not always obvious what extension operator to pick when rewriting an expression: A choice made at an outer level may affect the cost of rewriting deeply nested subexpressions. Perhaps this problem could be addressed using dynamic programming, but then the simplicity of rewriting would be lost.
- There is no obvious way to deal with garbage in the high bits. Attempts to introduce a gx ("garbage extend") operator lead to a warm feeling at first, but gx is a relation, not a function, and the math quickly becomes unwieldy.

Algebraic rewriting is still useful when dealing with the eight non-widenable operators, which perform bitwise rotations or overflow detection. Each application of a non-widenable operator must be translated, before widening, into an equivalent expression that uses only widenable operators. For example, the operator that determines if an addition overflows, add_overflows, can be rewritten using either of these equations:

$$
\begin{aligned}
\mathrm{add\_overflows}_n(x,y) &= \mathrm{ne}(\mathrm{sxlo}_{w\leftarrow w}(n, \mathrm{sx}_{w\leftarrow n}\, x + \mathrm{sx}_{w\leftarrow n}\, y), \mathrm{sx}_{w\leftarrow n}\, x + \mathrm{sx}_{w\leftarrow n}\, y) \\
\mathrm{add\_overflows}_n(x,y) &= \mathrm{add\_overflows}_w(\mathrm{shl}(\mathrm{sx}_{w\leftarrow n}\, x, w-n), \mathrm{shl}(\mathrm{sx}_{w\leftarrow n}\, y, w-n))
\end{aligned}
$$

where ne is inequality and shl is shift left.

Although the amount of machinery is surprising, our solution does provide a satisfying separation of concerns:

- The capabilities of the target machine are captured in the machine environment M.
- Global decisions about variables are captured in the fill-type environment Γ.
- Machine-independent truths about the widenability of operators are captured in their fill types (Fig. 4).

We hope that fill types, especially, will be of lasting value to anyone who has to think about running narrow programs on a wide machine.

Acknowledgements

This work has been funded in part by NSF grants CCR-0311482 and CCR-0096069. João Dias, Paul Govereau, Glenn Holloway, Reuben Olinsky, Sukyoung Ryu, and Chung-chieh Shan provided helpful comments on the paper.

Bibliography

David F. Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. In *Proceedings of the Joint ACM Java Grande/ISCOPE Conference*, pages 68–77, Stanford, California, June 2001.

Mihai Budiu, Majd Sakr, Kip Walker, and Seth C. Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the EuroPar 2000 European Conference on Parallel Computing*, volume 1900 of *LNCS*, pages 969–979, August 2000.

Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, Redwood City, CA, 1995.

Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Effective sign extension elimination. *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 37(5):187–198, May 2002.

Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. `C--`: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *LNCS*, pages 1–28. Springer Verlag, September 1999.

Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, volume 1474 of *LNCS*, pages 172–188. Springer Verlag, June 1998.

Norman Ramsey and Simon L. Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 35(5):285–298, May 2000.

Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80. IEEE/ACM, November 1994.

Kevin Redwine and Norman Ramsey. Widening integer arithmetic. Technical Report TR-01-04, Division of Engineering and Applied Sciences, Harvard University, January 2004.

Mark Stephenson, Jonathan Babb, and Saman P. Amarasinghe. Bitwidth analysis with application to silicon compilation. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 35(5):108–120, May 2000.