Reprinted from *Proceedings of the 2011 ACM Symposium on Principles of Programming Languages* **(POPL'11)**

Resourceable, Retargetable, Modular Instruction Selection Using a Machine-Independent, Type-Based Tiling of Low-Level Intermediate Code

Norman Ramsey
Department of Computer Science, Tufts University
nr@cs.tufts.edu

João Dias
Department of Computer Science, Tufts University
dias@cs.tufts.edu

Abstract

We present a novel variation on the standard technique of selecting instructions by tiling an intermediate-code tree. Typical compilers use a different set of tiles for every target machine. By analyzing a formal model of machine-level computation, we have developed a single set of tiles that is *machine-independent* while retaining the expressive power of machine code. Using this tileset, we reduce the number of tilers required from one per machine to one per architectural family (e.g., register architecture or stack architecture). Because the tiler is the part of the instruction selector that is most difficult to reason about, our technique makes it possible to retarget an instruction selector with significantly less effort than standard techniques. Retargeting effort is further reduced by applying an earlier result which generates the machine-dependent implementation of our tileset automatically from a declarative description of instructions' semantics. Our design has the additional benefit of enabling modular reasoning about three aspects of code generation that are not typically separated: the semantics of the compiler's intermediate representation, the semantics of the target instruction set, and the techniques needed to generate *good* target code.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation; D.3.4 [*Processors*]: Retargetable compilers

General Terms Algorithms, Theory

1. Introduction

Since compilers were first written, researchers have worked toward the goal of implementing N programming languages on M target machines with only $O(N+M)$ work instead of the $O(N \times M)$ work required to write a compiler for each language on each machine (Conway 1958; Strong et al. 1958). This goal has led to years of fruitful work on retargetable compilers, and at the current state of the art, the major $O(N \times M)$ component of a retargetable compiler is the *instruction selector*, which maps compiler-specific intermediate code to target-specific machine code. We have developed a new way of selecting instructions which reduces N to the number of compiler infrastructures and reduces M to the number of architectural families supported. Our implementation supports one compiler infrastructure and two architectural families (register machines and stack machines).

POPL'11, January 26–28, 2011, Austin, Texas, USA.

Reprinted from POPL'11,, [Unknown Proceedings], January 26–28, 2011, Austin, Texas, USA., pp. 575–586.

Our compiler infrastructure is built on C--, an abstraction that encapsulates an optimizing code generator so it can be reused with multiple source languages and multiple target machines (Peyton Jones, Ramsey, and Reig 1999; Ramsey and Peyton Jones 2000). C-- accommodates multiple source languages by providing two main interfaces: the C-- *language* is a machine-independent, language-independent target language for front ends; the C-- *run-time interface* is an API which gives the run-time system access to the states of suspended computations.

C-- is *not* a universal intermediate language (Conway 1958) or a "write-once, run-anywhere" intermediate language encapsulating a rigidly defined compiler and run-time system (Lindholm and Yellin 1999). Rather, C-- encapsulates compilation techniques that are well understood, but expensive to implement. Such techniques include instruction selection, register allocation, implementation of procedure calling conventions, instruction scheduling, scalar optimizations, and loop optimizations.

To accomodate a variety of source languages, the C-- language is very expressive. The implementation of a code generator for C-- therefore presents an unusual challenge: ensuring that the code generator accepts all valid input programs is harder than usual, and the compiler writer's job of mapping the language to target-machine code is bigger than usual. In this paper we present a new *tileset* for machine-level computation, which helps address both problems. Our work makes these contributions:

- We have developed a new variation on a standard model of machine-level computation: register transfers. By formalizing and analyzing register transfers, we have found a way of decomposing register transfers into *tiles*, where a tile represents a simple operation such as a sign-extending load or a three-register arithmetic operation (Section 5.2). The decomposition is the same for every target machine and is implemented once.
- By contrast with current techniques, which require a compiler writer to show that tiling is correct and complete for each new target machine, our techniques require the compiler writer to show correctness and completeness just once per architectural family. We argue informally that our tiling algorithm is correct and complete for the family of register machines (Section 6).
- We show that the syntax and type structure of register transfers, which are almost trivially simple, nevertheless provide enough structure around which to design an instruction selector.
- Finally, our design enables modular reasoning about a code generator that emits quality code; in particular, we decouple knowledge of the compiler's intermediate code from knowledge of the target machine, simplifying the task of retargeting the instruction selector to the point where most of the work can be done automatically (Dias and Ramsey 2010).

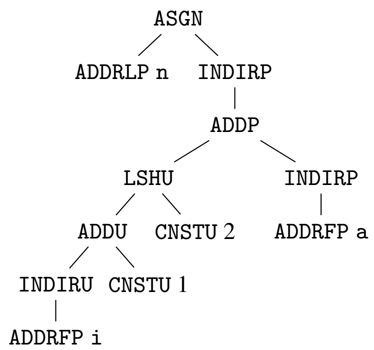

Figure 1. `lcc` IR for `n = a[i+1];`

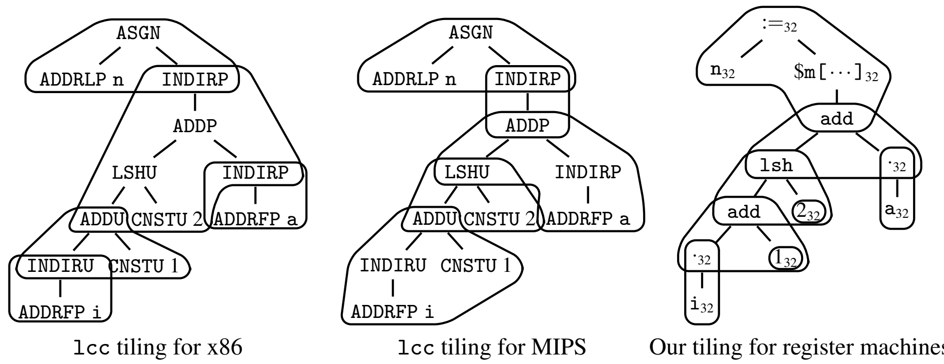

Figure 2. Multiple tilings of the tree from Figure 1

2. Background

The problem we solve is the mapping from a compiler's low-level intermediate code to target machine instructions: *instruction selection*. For a compiler that generates code for more than one target, instruction selection should be easily *retargetable*.

2.1 Related work: Instruction selection by tiling

Instruction selection is usually treated as a *tiling* problem. For each target machine, the compiler writer defines a set of *tiles*, where each tile maps a tree of the compiler's intermediate code (an *IR tree*) to an effective address or to one or more machine instructions. Using the tiles, the instruction selector covers the tree so that the root of one tile is a leaf of another tile, and so on. Such a covering is called a *tiling*. Postorder traversal of a tiling produces a sequence of instructions that implements the tree.

Figure 1 shows an IR tree created by the `lcc` compiler (Fraser and Hanson 1995) from the C assignment `n = a[i+1]`, where `i` is an unsigned integer, `n` is a signed integer, and `a` is a pointer to an array of signed integers. If you are not familiar with `lcc`'s IR, suffix `P` or `U` indicates the result is a pointer or an unsigned integer. `lcc` assumes that all variables are in memory, so a use of a variable is an `INDIR` node above an `ADDRF` (formal parameter) or `ADDRL` (local variable) node. A later pass may convert some `INDIR`/`ADDR` combinations to `REG`. The example tree also includes constants (`CNST`), operations (`ADD`, `LSH`), and an assignment (`ASGN`).

In a typical compiler, each target machine requires its own unique set of tiles or *tileset*. Figure 2 shows tilings of the tree in Figure 1 using three different tilesets. The left-hand tiling uses `lcc`'s x86 tileset. The middle tiling uses `lcc`'s MIPS tileset. The right-hand tiling uses our register-machine tileset, which works for both x86 and MIPS as well as other typical register machines such as ARM and PowerPC. (Our tiling, instead of `lcc`'s IR, shows the IR used by our Quick C-- compiler: the tree is the same, but the nodes are labeled differently. In particular, $\cdot_{32}$ labels a fetch from a variable.)

Given a tileset, there is often more than one way to tile a particular tree. The way is computed by a *tiling algorithm*, of which there are two common families: bottom-up rewriting and maximal munch.

Bottom-up rewriting, often called BURS or BURG, computes an optimal tiling. A bottom-up rewriter uses dynamic programming, and it is usually built using a code-generator generator: the compiler writer defines the tiles using a domain-specific language, and the code-generator generator uses the tiles to build a bottom-up tree matcher (Pelegrí-Llopart and Graham 1988; Aho, Ganapathi, and Tjiang 1989; Fraser, Henry, and Proebsting 1992; Proebsting 1992; Fraser, Hanson, and Proebsting 1992).

Maximal munch is a greedy tiling algorithm that works by top-down pattern-matching on the IR tree (Cattell 1980; Appel 1998). The tiling may not be optimal, but the algorithm is easy to implement by hand. Our compiler uses maximal munch.

2.2 Retargeting is surprisingly complicated

To the uninitiated, writing an instruction selector for a new machine seems simple: just define a tile for each machine instruction. But the task is more complicated than it seems:

- *A tile does not describe an instruction's semantics.* A tile might map a small IR tree to a machine instruction that implements the tree. But it is not safe to assume that the instruction performs *only* the computation described by the IR tree. For example, it is common to have a tile that maps an addition operation to a machine instruction that performs not only the addition but also an extra assignment to a condition-code register. To use such an instruction, the compiler writer must reason about whether the extra assignments are safe.
- *Having one tile for each machine instruction may not be sufficient to generate code.* The instruction selector must be able to tile *any* IR tree. Consequently, it is not possible to define a set of tiles without reasoning about what IR trees can be covered by the tiles. For example, on a 32-bit RISC machine, there is no instruction that loads a 32-bit immediate constant into a register. If the tileset contains only one tile for each machine instruction, the compiler will not be able to generate code for an IR tree that loads an immediate constant.

  The load-immediate problem is an instance of a general problem: to make sure that every IR tree can be covered, the compiler writer must define tiles that map to *sequences* of machine instructions, not just single instructions. To discover these tiles, it's not enough to think about the instruction set; the compiler writer has to think about the instruction set *and* about the structure of IR trees—and about what IR trees can occur during compilation.
- *Having one tile for each IR node may not be sufficient to generate* good *code.* To generate efficient code, you might need multiple tiles that can cover the same IR tree using different machine instructions. The x86 tiling on the left of Figure 2 shows two examples in which an addition is tiled using something other than the obvious `add` instruction. The small tile implementing `ADDU(INDIRU(···), CNSTU 1)` uses the x86 *load effective address* instruction, which has a small binary encoding and which doesn't affect condition codes. As another example, the largest tile, which performs a left shift, an add, and a fetch

(INDIRP), uses the indexed addressing mode to do the addition in the x86 addressing unit. This instruction not only has a small binary encoding but also frees up an integer unit for some other computation, enhancing instruction-level parallelism.

These complications arise because the standard approach to instruction selection, mapping IR fragments to machine instructions, inhibits modular reasoning: a compiler writer must reason about interactions among the compiler's IR, the target instruction set, and the optimizer. Such reasoning requires broad expertise. Worse, for each new machine, crafting new tiles requires that similar reasoning be repeated using somewhat different but equally broad expertise.

3. A modular approach to instruction selection

To reduce the effort required to write a code generator for a new target machine, we have developed a new, modular approach to instruction selection. Our approach separates concerns about the compiler from concerns about the machine and concerns about low-level optimization. We separate these concerns by refactoring the compiler in two ways: we optimize code *after* translating IR to machine instructions, and we divide tiling into separate machine-dependent and machine-independent components.

3.1 Related work: Optimizing machine codes

To separate the concerns of choosing machine instructions and generating efficient code, we adapt the approach developed by Davidson and Fraser (1984). In this approach, the compiler writer maps IR trees to *naïve* machine code, represented using *register-transfer lists*, also called RTLs (Section 4). RTLs are later improved by an optimizer. The optimizer is *machine-independent*, but it improves the program under the machine-dependent constraint that each RTL represents a single machine instruction: the *machine invariant*. The machine invariant is enforced by a predicate called the *recognizer*. Optimizing RTLs does not rule out other optimizations that operate on IR trees or on even higher-level representations, but Davidson has shown that many optimizations which are typically performed on IR are better performed on RTLs, where they help generate particularly efficient code (Benitez and Davidson 1988, 1994).

To ensure that the compiler emits efficient machine instructions, the optimizer includes a machine-independent peephole optimizer. Because the peephole optimizer translates the tiler's naïve machine code into efficient machine code, this way of organizing a compiler is sometimes called instruction selection by peephole optimization, and the peephole optimizer is sometimes called the instruction selector. This organization is used not only in Davidson's vpo compiler but also in gcc.

In Davidson and Fraser's original design, the mapping from low-level IR to RTLs satisfying the machine invariant is done by a component called the *code expander*. The expander is so named because it maps a typical IR tree to a long sequence of simple RTLs, expanding the code. Davidson and Fraser (1984) note that code expanders are simple and easy to to implement; to add a new machine, compiler writers are advised to clone and modify the expander for a similar machine. But since 1984, IRs, machines, and expanders have become more complex, and having an expander for each combination of source language and target machine creates maintenance problems: when bugs are discovered, there is not a single point of truth about how to do the translation. In our adaptation of the design, the code expander is split into three parts:

- A language-dependent part maps the compiler's low-level IR to RTLs that respect only the word size and byte order of the target machine—*not* the machine invariant.
- The tiler described in this paper expands these RTLs to RTLs that conform to the shapes described in Section 5.2 below.
- A machine-dependent implementation, which can be generated automatically (Dias and Ramsey 2010), computes a representation of each shape using RTLs that represent machine instructions—i.e., RTLs that respect the machine invariant.

We changed Davidson and Fraser's original design in a modest way: we took a single component and split it into three. This change, although modest, has a big impact: the number of components that have to be maintained drops from $N \times M$ to $N + M$, and the M machine-dependent components can be generated automatically from declarative descriptions of the target-machine semantics.

A big payoff of Davidson's approach is that responsibility for producing efficient machine code is isolated in the optimizer. The tiler is responsible not for performance but only for mapping IR trees to simple machine instructions. The naïveté of our tiler's output can be seen in Figure 2: compared to lcc's locally optimal, machine-dependent tilings, Quick C-- uses eight tiles instead of five. This naïve code is improved by the optimizer.

3.2 Modularity through machine-independent tiling of RTLs

Davidson's body of work shows that we can build an excellent compiler by first generating naïve machine code, then improving it. But even when we don't need to generate *good* machine code, writing a traditional tiler still requires that we reason simultaneously about the semantics of machine instructions and about the compiler's intermediate representation. In this paper, we show a novel alternative, which supports modular reasoning:

- Low-level IR may be translated to almost any well-typed RTLs. Unlike earlier compilers, which require that all RTLs satisfy the machine invariant, our compiler requires only that RTLs respect the byte order and word size of the target machine. This innovation frees the front end from having to reason about the target instruction set.
- Provided there is a machine-dependent recognizer, RTLs support machine-independent optimization. The recognizer decouples reasoning about *optimization* from reasoning about the instruction set. Dias and Ramsey (2006) show how to generate a recognizer from a description of instructions' semantics.
- RTLs are language-neutral: no matter what the source language, if the source code can be compiled to a low-level intermediate representation, that representation can easily be translated into RTLs. Reasoning about the source language is therefore confined to the front end, which emits RTLs.
- Part of our strategy of language-neutrality is to avoid high-level types. Instead, we use a type system that treats values only as bit vectors or Booleans, much as the hardware does (Section 4.1). This simple type system leads us to the centerpieces of our design: a single, machine-independent tileset that can be reused for any register machine (Section 5.2), and a tiling algorithm that reduces control-flow graphs to that tileset (Section 6).
- A compiler writer can reuse our tileset and tiling algorithm unchanged for any register machine. For each new target machine, all the compiler writer has to do is implement our machine-independent tileset using machine instructions. Our tiles are simple enough that this process can be automated by using a *tileset generator* (Dias and Ramsey 2010).

  Our tileset is *almost* target-neutral: while not suitable for *any* machine, it is suitable for any machine that provides roughly interchangeable *registers*. (We have defined another tileset for *stack* machines, but it has been used only to generate code for the x86 legacy floating-point unit.)

Using our modular approach, it is significantly easier to add a new target machine: because the tileset doesn't change, you don't have to design tiles, and you know our tileset will cover any IR tree. Even if you don't use our tileset generator, our experience shows that it is easy to implement the tileset by hand. Finally, our modular approach does not affect the code ultimately generated by the compiler; like any other Davidson/Fraser compiler, our compiler generates high-quality code if and only if the optimizer used after instruction selection is of high quality.

Using machine-independent intermediate forms to improve modularity is standard practice; prominent examples include the Java Virtual Machine (Lindholm and Yellin 1999) and LLVM (Lattner and Adve 2004). Some of these intermediate forms are very close to machine instructions. Our innovation is not that the intermediate form is machine-independent or low-level; rather, it is that

- As described below, the intermediate form is derived in a principled way from the type system of a metalanguage used to describe instructions' *semantics* (Ramsey and Davidson 1998).
- As described by Dias and Ramsey (2010), the intermediate form *eliminates* the need for the compiler writer to specify how each tile is implemented by machine instructions—the implementations can be discovered automatically by heuristic search.

Below, we present the ideas that lead to our design: we explain RTLs (Section 4); we discuss their static semantics (Section 4.1); we analyze their types (Section 4.2); we show how types suggest tiles (Section 5); and we present a tiling algorithm (Section 6).

4. Register-transfer lists and control-flow graphs

Register-transfer lists (RTLs) provide a simple but precise representation of the effects of machine instructions. A register-transfer list denotes a function from machine states to machine states. A machine state is represented as a collection of *storage spaces*, each of which we designate with a lower-case letter. For example, on the popular x86 architecture, we write `m` for memory, `r` for general-purpose registers, `f` for legacy floating-point registers, `x` for SSE registers (`%xmm`*n*), and `c` for control registers (including flags and the instruction pointer). Each storage space is divided into *cells*; a cell is the smallest natural unit with which the hardware reads and writes the storage. On the x86, for example, the cell size is 8 bits for memory, 32 bits for general-purpose registers, and 80 bits for legacy floating-point registers. (Access to a "register" like AL or AH is treated as access to an internal "slice" of a full cell like EAX.)

After program variables have been mapped to machine locations, RTLs are represented using the syntax shown in Figure 3.

- A register-transfer list is a list of guarded effects. Each effect represents the transfer of a value into a storage location, i.e., an assignment. The transfer occurs only if the guard (an expression) evaluates to true (**T**). Effects in a list take place simultaneously, as in Dijkstra's multiple-assignment statement; an RTL represents a single change of state. For example, an RTL can represent a swap instruction without introducing temporaries.
- A location may be a single cell or an aggregate of consecutive cells within a storage space. For example, four 8-bit bytes may be aggregated to form a location holding a 32-bit word, as in $\$\mathtt{m}_{\text{BIG},8}\,\mathtt{[}addr\mathtt{]}_{32}$, which specifies a big-endian word starting at address *addr*. In an aggregate, byte order is explicit.

  Some storage spaces stand for "temporary registers," or *temporaries*. Eventually the register allocator replaces temporaries with hardware registers (Dias and Ramsey 2006, §2.1).
- Values are computed by expressions that have no side effects. An expression may be a compile-time constant (k_w, **T**, or **F**), a link-time constant L, a fetch from a location l_w, or an application of an *RTL operator* $\oplus_\tau$ to a list of expressions.

a	Aggregation order (big-endian, little-endian, or ID)
c	Cell width of storage space
e, g	Expression (aka *guard*)
G	Control-flow graph
k	Literal integer or bit vector
l	Location
L	Label
$\oplus$	RTL operator
τ	Type
w	Width in bits

$$
\begin{array}{lcl}
RTL & \Rightarrow & \big[g \rightarrow \mathit{effect} \; \{ \mid g \rightarrow \mathit{effect} \} \big] \\
\mathit{effect} & \Rightarrow & l :=_w e \\
l & \Rightarrow & \$\mathit{space}_{a,c}\,\mathtt{[}e\mathtt{]}_w \\
e & \Rightarrow & k_w \mid \mathbf{T} \mid \mathbf{F} \mid L \mid l_w \mid \oplus_\tau(e_1, \ldots, e_n) \\
G & \Rightarrow & \varepsilon \mid RTL \mid G_1; G_2 \mid L\!: \mid \mathbf{goto}\; e \mid L_T \lhd g \rhd L_F
\end{array}
$$

Figure 3. Syntax of RTLs and control-flow graphs

RTLs are themselves composed into *control-flow graphs* G, whose syntax is also shown in Figure 3. Straight-line code includes the empty graph ε, a single RTL, or a sequence of graphs. Control flow is represented using labels and branches. (We take one liberty with the syntax; instead of writing $L\!:; G$, we write the clearer $L\!: G$.) For conditional branches, we use the nonstandard notation $L_T \lhd g \rhd L_F$, which means the same as "**if** g **then goto** L_T **else goto** L_F." This notation makes the inference rules in Section 6 easier to read.

As part of instruction selection, each conditional and unconditional branch in a control-flow graph is associated with a machine instruction, which is represented by an RTL that assigns to the program counter (PC). For example, a conditional branch is represented by an assignment like $\$\mathtt{r[4]} > 0 \rightarrow PC := L$. For computational instructions, the PC is updated implicitly, as formally specified elsewhere (Ramsey and Cifuentes 2003).

A control-flow graph can represent any code from a single statement all the way up to an entire procedure body. For example, on a 32-bit big-endian machine, the C statement `n = a[i+1]` could be translated to a flow graph consisting of the RTL

$$\mathtt{n}_{32} :=_{32} \$\mathtt{m}_{\text{BIG},8}\,\mathtt{[add}_{\tau 32}(\mathtt{a}_{32}, \mathtt{shl}_{\tau 32}(\mathtt{add}_{\tau 32}(\mathtt{i}_{32}, 1_{32}), 2_{32}))\mathtt{]}_{32}$$

where $\tau 32 = 32\;\mathtt{bits} \times 32\;\mathtt{bits} \rightarrow 32\;\mathtt{bits}$.

In contrast to RTLs used in some earlier work, C--'s RTLs have a precise semantics independent of any particular machine. The details are not relevant to the contribution of this paper, but you can imagine the simplest possible denotational semantics:

- A syntactic location l denotes a function from a machine state to a hardware location holding a bit vector. The location may be a single cell or a sequence of cells in a storage space.
- A syntactic expression e denotes a function from a machine state to a value, which is either a Boolean or a bit vector.
- A syntactic effect or a syntactic RTL denotes a function from a machine state to another machine state. If two effects in the same RTL mutate the same cell, the parallel composition of those effects denotes the constant wrong function ($\perp$). Denotations are strict, so if a computation goes wrong, it stays wrong.
- An RTL operator denotes a pure function on values.

As usual, the dynamic semantics describes the observable effects of a program. These effects must be preserved by the tiling algorithm, just as by any other instruction selector. The algorithm, however, is inspired not by the dynamic semantics but by the static semantics.

n	A width variable
w	A literal width
w_M	The width of an address on the target machine
n `bits`	A value that is n bits wide
n `loc`	A location containing an n-bit value
`bool`	A Boolean condition

$\boxed{\Delta \vdash_\tau \tau}$ (τ is a well-formed type)

$$\frac{n \in \Delta}{\Delta \vdash_\tau n\,\texttt{bits}} \qquad \frac{n \in \Delta}{\Delta \vdash_\tau n\,\texttt{loc}} \qquad \frac{w \in \mathbb{N}}{\Delta \vdash_\tau w\,\texttt{bits}} \qquad \frac{w \in \mathbb{N}}{\Delta \vdash_\tau w\,\texttt{loc}}$$

$$\frac{}{\Delta \vdash_\tau \texttt{bool}} \qquad \frac{\Delta, n \vdash_\tau \tau}{\Delta \vdash_\tau \forall n.\tau} \qquad \frac{\Delta \vdash_\tau \tau_i \qquad \Delta \vdash_\tau \tau}{\Delta \vdash_\tau \tau_1 \times \ldots \times \tau_n \to \tau}$$

Figure 4. Types and type-formation rules

4.1 Static semantics of RTLs

Our RTLs are typed, but not as expressively as typed intermediate languages or assembly languages. Our type system tells us how *wide* something is—how many bits are in a value or a location—and whether an expression computes a bit vector or a Boolean.

Our type system is inspired by System F (Girard 1986), but it is both extended and restricted:

- Our system is extended by allowing a type to be quantified over an *integer*. The integer is a *width* and tells how many bits are in a value or in a location.
- Our system is restricted in ways that might remind you of the Hindley-Milner system (Milner 1978): The only polymorphic types are prenex-quantified type schemes, and only RTL operators, which are named and bound in the environment Γ_0, may have polymorphic types. At each use, a type scheme is fully instantiated, so the type of any *term* in the RTL language (from Figure 3) is always monomorphic.

Types and type-formation rules are shown in Figure 4. Rules for typing operators, locations, and expressions are shown in Figure 5. The side condition "w_s indexes s" says that storage space s is indexed by an an address or register number that is w_s bits wide. If s refers to main memory, then w_M indexes s.

By giving two distinct rules for literal constants k, we emphasize a design decision: unlike many other intermediate representations, our RTLs do not distinguish signed integers, unsigned integers, floating-point numbers, and pointers. In this paper, we pretend k is an integer, and we show two side conditions under which an integer k can be said to fit in w bits. In our implementation, a compile-time constant k has an abstract type, and the side conditions are imposed by the operations used to create values of this type.

The important part of the type system is the specialization of width-polymorphic RTL operators, as shown in the judgment form $\Gamma_0 \vdash_{op} \oplus : \tau$. In this judgment, Γ_0 represents the collection of operators defined in the C-- language specification (Ramsey, Peyton Jones, and Lindig 2005). This collection represents a *union machine*: it includes any operator we think might be used to describe a machine instruction on a current or future target. For example, the collection includes `popcnt`, which returns the number of 1 bits in a word. This operator is native to Intel architectures; on other machines, we use algebraic laws to rewrite it as a composition of other operators, much in the spirit of Warren (2003). Thinking about all the machine-level operators we ever heard of—and their types—led us to our new tiling for code generation.

$\boxed{\vdash l : \tau}$

$$\frac{\vdash e : w_s\,\texttt{bits} \qquad w_s \text{ indexes } s \qquad c = w}{\vdash \$s_{\textsc{id},c}\,[e]_w : w\,\texttt{loc}} \qquad \frac{\vdash e : w_s\,\texttt{bits} \qquad w_s \text{ indexes } s \qquad a \in \{\textsc{Big}, \textsc{Little}\} \qquad c \text{ divides } w}{\vdash \$s_{a,c}\,[e]_w : w\,\texttt{loc}}$$

$\boxed{\vdash e : \tau}$

$$\frac{0 \le k < 2^w}{\vdash k_w : w\,\texttt{bits}} \qquad \frac{-2^{w-1} \le k < 2^{w-1}}{\vdash k_w : w\,\texttt{bits}} \qquad \frac{}{\vdash L : w_M}$$

$$\frac{\vdash l : w\,\texttt{loc}}{\vdash l_w : w\,\texttt{bits}} \qquad \frac{\Gamma_0 \vdash_{op} \oplus : \tau_1 \times \ldots \times \tau_n \to \tau \qquad \vdash e_i : \tau_i}{\vdash \oplus_{\tau_1 \times \ldots \times \tau_n \to \tau}(e_1, \ldots, e_n) : \tau}$$

$\boxed{\Gamma_0 \vdash_{op} \oplus : \tau}$ (operator $\oplus$ can be instantiated at type τ)

$$\frac{\oplus \in \operatorname{dom} \Gamma_0}{\Gamma_0 \vdash_{op} \oplus : \Gamma_0(\oplus)} \qquad \frac{\Gamma_0 \vdash_{op} \oplus : \forall n.\tau}{\Gamma_0 \vdash_{op} \oplus : \tau[n \mapsto w]}$$

Figure 5. Typing rules for locations and selected expressions

4.2 Classifying RTL operators by type

Although the C-- language specification lists 81 RTL operators (28 floating-point operators and 53 integer and bitwise operators), these operators have only 14 distinct types (Table 6). Before explaining how these types help classify RTLs (Section 5 below), we put the 14 types into just 5 groups:

- A *standard value operator* takes some bit vectors of reasonable width and returns a result of that width. (The width is typically the width of one word on the target machine.) These operators are summarized in the first two rows of Table 6. Binary operators include two's-complement addition and subtraction, signed and unsigned division, quotient and remainder, bitwise Boolean operations, and rotations and shifts. Unary operators include bitwise complement, two's-complement negation, population count, and floating-point absolute value and negation.
- A *weird value operator* takes some bit vectors of reasonable width, but also takes an argument or returns a result of another ("weird") width. Such operators require special treatment during tiling. Most floating-point operators are weird value operators because in addition to their ordinary operands, they take a 2-bit rounding mode. The other weird value operators are extended multiplies (which double the width of their operands), nullary rounding-mode "operators" like `round_down`, and multiprecision operators such as carry, borrow, add with carry, and add with borrow.
- A *size-changing operator* widens or narrows a bit vector. The integer size-changing operators are sign extension (`sx`), zero extension (`zx`), and extraction of least-significant bits (`lobits`). Conversions between integers and floating-point values, and between floating-point values of different sizes (`f2f`), are also size-changing operators. Size-changing operators help formalize machine instructions, such as sign-extending loads, that convert values between representations of different sizes.
- A *comparison operator* takes two bit vectors and returns a Boolean. Comparison operators include the usual integer and floating-point comparisons.
- A *Boolean operator* takes one or more Booleans and returns a Boolean. The Boolean operators are `conjoin`, `disjoin`, and `not`.

These groups provide a starting point from which to classify RTLs.

5. From graphs to tiles via types

Before explaining how groups of operators influence the design of our tileset, we consider how to tile an arbitrary control-flow graph using a small set of tiles. As explained in Section 6.1 below, our tiling algorithm reduces every control-flow graph to a composition of graphs in which each node is either a single assignment, an unconditional branch, or a conditional branch on the results of a comparison operator. But even a single assignment or branch can contain an arbitrarily large expression on the right-hand side. To tile these nodes, we use a restricted subset of graphs, where the restrictions are motivated by common properties of register machines (Section 5.1). We then build on the groups of operators identified above to classify the restricted graphs and to identify tiles, using a new idea we call *shape* (Section 5.2).

5.1 Restricting graphs in the tileset

We would like as few graphs as possible to be tiles, because the fewer tiles there are and the simpler the tiles are, the easier it is to implement the tileset. But enough graphs need to be tiles to cover any single assignment or branch. In particular, because every RTL operator can appear in either an assignment or a branch, we need at least one tile for every RTL operator. We have designed our tileset around a fundamental assumption about register machines: if the target machine can compute operator $\oplus$, then the machine can apply $\oplus$ to arguments in registers, and it can place the result in a register (or if the result is Boolean, it can use the result to determine control flow).

We get our tiles by restricting the graphs and RTLs from Figure 3:

- A graph must be a single RTL, a label, **goto** L, **goto** t, or a conditional branch.
- An RTL must have exactly one assignment,[1] and its guard must be the literal **T**.
- If a graph or RTL refers to a memory location, the location's address must be stored in a register. (A memory location is one in which the actual location can be computed at run time or at link time. A register is a location that must be encoded in the instruction word at compile time.)
- The arguments of an RTL operator must be in locations, and except for size-changing operators, those locations must be registers. (Because putting a "weird" value into a register can be expensive, we also permit that a weird argument may be a compile-time constant.)

Any single assignment or branch can be tiled using these restricted graphs. For example (leaving widths and byte order implicit),

$$r_1 := r_2 + (r_3 \times 12)$$

can be reduced to this sequence of tiles:

$$\begin{aligned} t_1 &:= 12 \\ t_2 &:= r_3 \times t_1 \\ r_1 &:= r_2 + t_2. \end{aligned}$$

The reduction introduces fresh temporaries t_1 and t_2 to hold the values of subexpressions of the original RTL.

5.2 Using types to give shapes to tiles

The restrictions above still permit the formation of more than 90 tiles. To simplify both the explanation and the implementation of our tileset, we group tiles into equivalence classes we call *shapes*.

[1] We except procedure calls (not otherwise covered in this paper), which both capture and update the program counter, requiring two assignments.

Type of operator	*Number of operators*	*Sample*
$\forall n.n$ `bits` $\times\, n$ `bits` $\to n$ `bits`	17	`add`
$\forall n.n$ `bits` $\to n$ `bits`	5	`com`
$\forall n.n$ `bits` $\times\, n$ `bits` $\times\, 1$ `bits` $\to n$ `bits`	2	`addc`
$\forall n.n$ `bits` $\times\, n$ `bits` $\times\, 1$ `bits` $\to 1$ `bits`	2	`carry`
$\forall n.n$ `bits` $\times\, n$ `bits` $\times\, 2$ `bits` $\to n$ `bits`	4	`fadd`
$\forall n.n$ `bits` $\times\, 2$ `bits` $\to n$ `bits`	1	`fsqrt`
$\forall n.n$ `bits` $\times\, n$ `bits` $\to 2n$ `bits`	3	`fmulx`
$\forall n.n$ `bits`	4	`mzero`
2 `bits`	4	`round_down`
$\forall n,m.n$ `bits` $\to m$ `bits`	5	`sx`
$\forall n,m.n$ `bits` $\times\, 2$ `bits` $\to m$ `bits`	3	`f2f`
$\forall n.n$ `bits` $\times\, n$ `bits` $\to$ `bool`	24	`eq`
`bool` $\times$ `bool` $\to$ `bool`	2	`conjoin`
`bool` $\to$ `bool`	1	`not`

Table 6. Types of selected RTL operators

t	Any temporary or hardware register
m	Any hardware memory space
k	Any literal integer, bit vector, or other compile-time constant
L	Any label or other link-time constant
$\oplus$	An RTL value operator
r	A hardware register
rm	A floating-point rounding mode
$\omega, \hat{r}, \hat{t}$	A weird value, like a carry bit, in all or part of a register
$?$	Any RTL comparison operator

Table 7. Notational conventions for metavariables

A shape is characterized by a control-flow graph in which locations, literals, and operators may be represented by the metavariables in Table 7. Shapes for a register machine, which are inspired both by our restrictions on graphs and RTLs and by our grouping of RTL operators by type, are shown in Figure 8. The graphs with these shapes constitute our *tileset*. (We have also defined shapes for stack machines, such as the x86 floating-point unit; these shapes are sketched in Appendix A.) To simplify the presentation, we omit nonlocal control constructs such as call and return. We also omit some shapes that involve sign-extending or zero-extending a narrow result ω instead of storing it in a narrow location like ω_1.

A tile is a control-flow graph that can be obtained by substituting for metavariables in the tile's shape: we replace the metavariable $\oplus$ (if any) with an actual RTL operator of an appropriate type, and we replace metavariables r and t by hardware registers or temporaries appropriate to the context in which they appear. For example, given the `binop` shape $t := \oplus(t_1, t_2)$, we get a shift-left tile by using `shl` in place of $\oplus$ and by using temporaries that stand for general-purpose registers. As another example, the `store` tile might require an address or integer register for temporary t_1 but should support any register t as the value to be stored.

5.3 Tiling through a code-generation interface

Each shape in Figure 8 is labeled with a name. The names identify functions in our *code-generation interface*, which encapsulates a machine-dependent implementation of every tile. Our instruction selector works as follows:

- The tiler expects an input control-flow graph in which each node contains a well-typed RTL. Moreover, the RTLs in an input graph may contain only trivially true guards. (The C-- language cannot express nontrivial guards.) Nontrivial Boolean

Standard and weird value operators

$t := \oplus(t_1, t_2)$	`binop`	(a binary ALU operation)
$t := \oplus(t_1)$	`unop`	(a unary ALU operation)
$t := \oplus(t_1, t_2, rm)$	`binrm`	(a binary floating-point operation)
$t := \oplus(t_1, rm)$	`unrm`	(a unary floating-point operation or conversion)
$t := \oplus(t_1, t_2, \omega)$	`wrdop`	(a weird value operation, like add with carry)
$\omega_1 := \oplus(t_1, t_2, \omega_2)$	`wrdrop`	(a weird value/result operation, like carry)
$t_{hi}, t_{lo} := t_1 \otimes t_2$	`dblop`	(an extended multiply operation)

Size-changing operators

$t := \mathtt{sx}(m[t_1])$	`sxload`	(load signed byte/halfword)
$t := \mathtt{zx}(m[t_1])$	`zxload`	(load unsigned byte/halfword)
$m[t_1] := \mathtt{lobits}_n(t)$	`lostore`	(store byte/halfword)

Data movement (most tiles transfer exactly one word)

$m[t_1] := t$	`store`	
$t := m[t_1]$	`load`	
$m[t_1] := m[t_2]$	`block_copy`	(any number of bytes)
$t_1 := t_2$	`move`	
$t := \omega$	`hwget`	
$\omega := t$	`hwset`	
$t := k$ or $t := L$	`li`	(load immediate constant)

Control flow (including comparison operators)

goto L	`b`	(branch)
goto t	`br`	(branch register)
$L_T \triangleleft t_1 ? t_2 \triangleright L_F ; L_F$:	`bc`	(branch conditional)

Figure 8. Shapes of tiles for a register machine

expressions g appear only in conditional branches $L_T \triangleleft g \triangleright L_F$. (If a target machine supports guarded assignments, aka "predicated instructions," the proper RTLs can be found by standard optimizations *after* instruction selection.)

- The tiler covers each input graph with tiles. For each tile, the tiler identifies the shape, then calls the function in the code-generation interface which implements that shape. The tiler passes arguments which correspond to the metavariables shown in Figure 8; metavariables typically identify locations and an operator. The code-generation function returns a machine-dependent control-flow graph that implements the tile. This graph may contain RTLs with nontrivial guards.

  For example, to get a control-flow graph implementing the tile $r_1 := r_2 + t_2$, the tiler would call

  ```
  CG.binop ('r',1) ("add",[32]) ('r',2) ('t',2)
  ```

  The `binop` function would return a small control-flow graph, probably containing one RTL representing an add instruction. (In the example, the `[32]` in `("add",[32])` is the list of widths with which the polymorphic operator `add` is instantiated.)

- The tiler produces an output control-flow graph in which each node contains a well-typed RTL that is representable by a single instruction on the target machine.

The code-generation interface also associates each operator with a machine-dependent *context*, which tells the tiler what registers and temporaries may be operands or results of that operator.

5.4 Implementation of tilesets

The functions in the code-generation interface must be implemented for every target machine. Each function corresponds to a shape and must implement all tiles of that shape. The implementation of a tile may have side effects not called for in the interface, but these effects must be limited to *scratch registers*—which can't be named by any program—and to compiler temporaries. The implementor of the tileset decides which registers are scratch registers.

It is not difficult to implement a tileset by hand, but we have developed an algorithm for automatically generating a tileset from a declarative machine description (Dias and Ramsey 2010). Although the problem is undecidable in principle, our tileset generator produces complete tilesets for the x86, PowerPC, and ARM.

6. Specification of a machine-independent tiler

Our tiler takes as input a control-flow graph G' and returns a new graph G, which has the same observable effect as G', but which is composed entirely of RTLs that represent machine instructions. As shown in the example tiling in Section 5.1, RTLs in G may have side effects on locations not mentioned in G'. We therefore write the tiling transformation using the judgment form

$$G \overset{\mathscr{L}}{\subseteq} G',$$

which states that executing control-flow graph G has the same effect as executing graph G', except that G may also overwrite any location in the set $\mathscr{L}$. As shorthand, we say that G *implements* G'.

Graph G and set $\mathscr{L}$ satisfy important properties: G is made from tiles; $\mathscr{L}$ contains all the locations assigned by G, which we write $defs(G)$; and $defs(G)$ includes $defs(G')$. Any locations in $\mathscr{L}$ that are *not* in $defs(G')$ must be scratch registers or compiler temporaries, which cannot be observed by a program, so that G and G' have the same observable effect. These properties are established by metatheoretic reasoning.

We represent calls to the code-generation interface by a very similar judgment: $G \subseteq^{\mathscr{L}}{}_M\, G'$ says that graph G implements graph G', and moreover, every RTL in G is implementable by a single instruction on the target machine M. The $\subseteq^{\mathscr{L}}{}_M$ relation is significant for another reason: judgment $G \subseteq^{\mathscr{L}}{}_M\, G'$ appears only when G' is a tile.

The tiler satisfies two properties:

- *Soundness*: If $G \subseteq^{\mathscr{L}} G'$, then G is equivalent to G', modulo assignments to unobservable locations in $\mathscr{L}$.
- *Machine invariant*: If $G \subseteq^{\mathscr{L}} G'$, then G contains only RTLs implementable on the target machine.

These properties follow by inspection of the rules. Ideally the tiler would satisfy a third property:

- *Completeness*: If G' is composed of well-typed RTLs that use operators only at the word size of the target machine,[2] then there exist a G and an $\mathscr{L}$ such that $G \subseteq^{\mathscr{L}} G'$.

Like most compiler writers, we don't prove that our code generator is complete. Instead, we follow typical best practices: we reason informally about the completeness of the tiler, and we supplement that reasoning with regression tests. One advantage of our approach is that we have to do the reasoning just *once*, and the results apply to any register machine. The current state of the art is that the completeness of an instruction selector must be established once *per target machine*.

[2] The restriction on widths of integer and bitwise operators can be relaxed to permit values smaller than a machine word (Redwine and Ramsey 2004).

Par

$$\frac{G_1 \overset{\mathscr{L}_1}{\subseteq} l_1 := e_1 \qquad \forall i > 1 : \mathscr{L}_1 \parallel \mathit{uses}(e_i) \\ G_2 \overset{\mathscr{L}_2}{\subseteq} l_2 := e_2 \mid \cdots \mid l_n := e_n \qquad \mathscr{L}_2 \parallel \{l_1\}}{G_1; G_2 \overset{\mathscr{L}_1 \cup \mathscr{L}_2}{\subseteq} l_1 := e_1 \mid \cdots \mid l_n := e_n}$$

BreakCycle

$$\frac{t\ \texttt{fresh} \qquad G \overset{\mathscr{L}}{\subseteq} t := e_1 \mid l_2 := e_2 \mid \cdots \mid l_n := e_n \;;\; l_1 := t}{G \overset{\mathscr{L}}{\subseteq} l_1 := e_1 \mid \cdots \mid l_n := e_n}$$

Seq

$$\frac{G_1 \overset{\mathscr{L}_1}{\subseteq} G_1' \qquad G_2 \overset{\mathscr{L}_2}{\subseteq} G_2' \\ \mathscr{L}_1 \setminus \mathit{defs}(G_1') \parallel \mathit{uses}(G_2') \qquad \mathscr{L}_2 \parallel \mathit{defs}(G_1') \setminus \mathit{defs}(G_2')}{G_1; G_2 \overset{\mathscr{L}_1 \cup \mathscr{L}_2}{\subseteq} G_1'; G_2'}$$

Figure 9. Rules to eliminate parallel assignments and sequences

Techniques for proving completeness would represent a significant advance over the state of practice. An even greater advance would be to formalize the idea of architectural family and to prove completeness for an entire family. Both problems are beyond the scope of this paper.

The rest of this section presents rules for the tiling transformation. Although widths and byte order play a role in the tiling, they complicate the exposition of the algorithm, so we omit them except when they are crucial. This treatment is consistent with the treatment of widths and byte order in the C-- language itself: a C-- compiler infers widths when possible, so they are rarely needed in surface syntax. Widths are also inferred in our machine descriptions (Ramsey and Davidson 1998). Similarly, byte order is fixed for each storage space and so is inferred from the space.

6.1 Reducing graphs to RTLs

A single assignment can be covered by tiles whose shapes are given in Figure 8. But as noted in Section 4, an RTL can contain more than one assignment: to be able to express the semantics of machine instructions, RTLs must be able to express parallel assignments. In C--, we have chosen to make that expressive power available to front ends. Our compiler must therefore handle parallel assignments.

One way to handle a parallel assignment is to present it to the machine-dependent recognizer; if the RTL is accepted, it satisfies the machine invariant and it need not be tiled. If the recognizer does *not* accept a parallel assignment, we use the rules in Figure 9.

Rule Par takes a parallel assignment, extracts a single assignment $l_1 := e_1$, and executes it before the remaining assignments. This transformation preserves semantics only if an important side condition holds: for all $i > 1$, the value of e_i does not depend on the contents of l_1. The rule uses the slightly stronger condition $\mathscr{L}_1 \parallel \mathit{uses}(e_i)$, where writing $\parallel$ between two sets means they are disjoint. The stronger condition requires in addition that graph G_1, which computes $l_1 := e_1$, does not reuse temporaries that appear free in e_i. There is an additional side condition that graph G_2 does not change the value of l_1.

Because parallel composition of assignments is associative and commutative, rule par can be applied to a parallel assignment as long as some location l_i can be changed without affecting the value of any other expression e_j, $j \neq i$. Otherwise, all assignments have cyclic data dependencies, and we apply the BreakCycle rule, which breaks a cycle by introducing a fresh temporary.

Finally, the seq rule shows that graphs in a sequence are tiled independently. This rule expresses a standard code-generation technique: additional locations modified by G_1 (beyond what G_1' modifies) may not affect values computed in G_2, and similarly G_2 may not overwrite any observable locations modified by G_1.

6.2 Rules for value operators

To tile a single assignment of a value operator (Figure 10), we use the first group of shapes in Figure 8. The binop rule is representative; its task is to tile the assignment $t := \oplus(e_1, e_2)$. The first step is to allocate fresh temporaries to hold the values of e_1 and e_2 and to tile those RTLs recursively, producing graphs G_1 and G_2. The side condition $\mathscr{L}_1 \parallel \mathit{uses}(e_2)$ ensures that graph G_1 does not mutate any location on which the value of e_2 depends. Then, the judgment $G \subseteq^{\mathscr{L}}{}_M\ t := \oplus(t_1, t_2)$ calls the code-generation interface to get a machine-dependent implementation of the binary operator $\oplus$. The result of the tiling is the sequence $G_1; G_2; G$. All of the rules in Figure 10 work along similar lines, but a few are noteworthy.

In rules BinRM through WrdropZx, we use hat signs to indicate "weird" values, that is, values that aren't the same width as a register. For example, in the BinRM rule, expression $\hat{e}_3$ denotes an IEEE floating-point rounding mode (RM), which is only two bits wide. The hatted temporary $\hat{t}_3$ means something a little different: $\hat{t}_3$ stands for a standard register containing the value of $\hat{e}_3$ in its least significant two bits. In the weird-operator rules, one-bit carries and borrows are treated similarly. (Our tiler does some extra work not shown in the rules: it keeps track of whether the high bits of register $\hat{t}_3$ are zero, are copies of bit 1, or are unknown. Our tiler uses this information to tile sign-extension and zero-extension operations more efficiently.)

The last noteworthy rule in Figure 10 is the DblOp rule, which handles extended-multiply operators. An extended multiply takes two operands of equal width and produces a result of twice that width. In hardware, that result is expected to be split over two registers, as enforced by the code-generation interface in the judgment $G_3 \subseteq^{\mathscr{L}_3}{}_M\ t_{hi}, t_{lo} := \otimes(t_1, t_2)$. In C--, however, it is not possible to express an assignment of a single result to a register pair, so in a well-typed RTL, the assignment must be to memory. The addresses of the most and least significant words (*hiaddr* and *loaddr*) depend on byte order.

6.3 Rules for tiling size-changing operators

To tile a single assignment of a size-changing operator, we use the second group of shapes in Figure 8. We show rules for integer size-changing operators only; there are similar rules for changing the sizes of floating-point values and for converting between integer and floating point. Our intermediate language uses integer size-changing operators in two contexts: when transferring bytes (or halfwords, etc.) between registers and memory (Figure 11), and when transferring small values between special-purpose hardware registers and ordinary registers (Figure 12). Both groups of rules use the same techniques: when transferring a value from a narrow location to a wide location, the value must be sign-extended or zero-extended; when transferring a value from a wide location to a narrow location, only the least significant n bits are transferred. Both groups of rules allocate fresh temporaries to hold the values of subexpressions, which are tiled recursively.

6.4 Rules for data-movement tiles

To tile a single, data-movement assignment (Figure 13), we use the third group of shapes in Figure 8. Rules Store and Load

BINOP

$$\frac{\begin{array}{c} t_1,t_2\ \texttt{fresh} \\ \oplus:\forall n.n\ bits \times n\ bits \to\ n\ bits \qquad \mathscr{L}_1 \parallel uses(e_2) \\ G_1 \overset{\mathscr{L}_1}{\subseteq} t_1 := e_1 \qquad G_2 \overset{\mathscr{L}_2}{\subseteq} t_2 := e_2 \qquad G \overset{\mathscr{L}}{\subseteq}_M t := \oplus(t_1,t_2) \end{array}}{G_1;G_2;G \overset{\mathscr{L}\cup\mathscr{L}_1\cup\mathscr{L}_2}{\subseteq} t := \oplus(e_1,e_2)}$$

UNOP

$$\frac{\begin{array}{c} t'\ \texttt{fresh} \\ \oplus:\forall n.n\ bits \to\ n\ bits \qquad G \overset{\mathscr{L}}{\subseteq} t' := e \qquad G' \overset{\mathscr{L}'}{\subseteq}_M t := \oplus(t') \end{array}}{G;G' \overset{\mathscr{L}\cup\mathscr{L}'}{\subseteq} t := \oplus(e)}$$

BINRM

$$\frac{\begin{array}{c} t_1,t_2,\hat{t}_3\ \texttt{fresh} \\ \oplus:\forall n.n\ bits \times n\ bits \times 2\ bits \to\ n\ bits \qquad G_1 \overset{\mathscr{L}_1}{\subseteq} t_1 := e_1 \\ G_2 \overset{\mathscr{L}_2}{\subseteq} t_2 := e_2 \qquad G_3 \overset{\mathscr{L}_3}{\subseteq} \hat{t}_3 := \hat{e}_3 \qquad G \overset{\mathscr{L}}{\subseteq}_M t := \oplus(t_1,t_2,\hat{t}_3) \\ \mathscr{L}_1 \parallel uses(e_2) \cup uses(\hat{e}_3) \qquad \mathscr{L}_2 \parallel uses(\hat{e}_3) \end{array}}{G_1;G_2;G_3;G \overset{\mathscr{L}\cup\mathscr{L}_1\cup\mathscr{L}_2\cup L_3}{\subseteq} t := \oplus(e_1,e_2,\hat{e}_3)}$$

UNRM

$$\frac{\begin{array}{c} t_1,\hat{t}_2\ \texttt{fresh} \\ \oplus:\forall n.n\ bits \times 2\ bits \to\ n\ bits \qquad \mathscr{L}_1 \parallel uses(\hat{e}_2) \\ G_1 \overset{\mathscr{L}_1}{\subseteq} t_1 := e_1 \qquad G_2 \overset{\mathscr{L}_2}{\subseteq} \hat{t}_2 := \hat{e}_2 \qquad G \overset{\mathscr{L}}{\subseteq}_M t := \oplus(t_1,\hat{t}_2) \end{array}}{G_1;G_2;G \overset{\mathscr{L}\cup\mathscr{L}_1\cup\mathscr{L}_2}{\subseteq} t := \oplus(e_1,\hat{e}_2)}$$

WRDOP

$$\frac{\begin{array}{c} t_1,t_2,\hat{t}_3\ \texttt{fresh} \\ \oplus:\forall n.n\ bits \times n\ bits \times 1\ bit \to\ n\ bits \qquad G_1 \overset{\mathscr{L}_1}{\subseteq} t_1 := e_1 \\ G_2 \overset{\mathscr{L}_2}{\subseteq} t_2 := e_2 \qquad G_3 \overset{\mathscr{L}_3}{\subseteq} \hat{t}_3 := \hat{e}_3 \qquad G \overset{\mathscr{L}}{\subseteq}_M t := \oplus(t_1,t_2,\hat{t}_3) \\ \mathscr{L}_1 \parallel uses(e_2) \cup uses(\hat{e}_3) \qquad \mathscr{L}_2 \parallel uses(\hat{e}_3) \end{array}}{G_1;G_2;G_3;G \overset{\mathscr{L}\cup\mathscr{L}_1\cup\mathscr{L}_2\cup L_3}{\subseteq} t := \oplus(e_1,e_2,\hat{e}_3)}$$

WRDROPSX

$$\frac{\begin{array}{c} t_1,t_2,\hat{t}_3\ \texttt{fresh} \qquad \oplus:\forall n.n\ bits \times n\ bits \times 1\ bit \to\ 1\ bit \\ G_1 \overset{\mathscr{L}_1}{\subseteq} t_1 := e_1 \qquad G_2 \overset{\mathscr{L}_2}{\subseteq} t_2 := e_2 \\ G_3 \overset{\mathscr{L}_3}{\subseteq} \hat{t}_3 := \hat{e}_3 \qquad G \overset{\mathscr{L}}{\subseteq}_M \hat{t} := \texttt{sx}(\oplus(t_1,t_2,\hat{t}_3)) \\ \mathscr{L}_1 \parallel uses(e_2) \cup uses(\hat{e}_3) \qquad \mathscr{L}_2 \parallel uses(\hat{e}_3) \end{array}}{G_1;G_2;G_3;G \overset{\mathscr{L}\cup\mathscr{L}_1\cup\mathscr{L}_2\cup\mathscr{L}_3}{\subseteq} \hat{t} := \texttt{sx}(\oplus(e_1,e_2,\hat{e}_3))}$$

WRDROPZX

$$\frac{\begin{array}{c} t_1,t_2,\hat{t}_3\ \texttt{fresh} \qquad \oplus:\forall n.n\ bits \times n\ bits \times 1\ bit \to\ 1\ bit \\ G_1 \overset{\mathscr{L}_1}{\subseteq} t_1 := e_1 \qquad G_2 \overset{\mathscr{L}_2}{\subseteq} t_2 := e_2 \\ G_3 \overset{\mathscr{L}_3}{\subseteq} \hat{t}_3 := \hat{e}_3 \qquad G \overset{\mathscr{L}}{\subseteq}_M \hat{t} := \texttt{zx}(\oplus(t_1,t_2,\hat{t}_3)) \\ \mathscr{L}_1 \parallel uses(e_2) \cup uses(\hat{e}_3) \qquad \mathscr{L}_2 \parallel uses(\hat{e}_3) \end{array}}{G_1;G_2;G_3;G \overset{\mathscr{L}\cup\mathscr{L}_1\cup\mathscr{L}_2\cup\mathscr{L}_3}{\subseteq} \hat{t} := \texttt{zx}(\oplus(e_1,e_2,\hat{e}_3))}$$

DBLOP

$$\frac{\begin{array}{c} t_1,t_2,t_{lo},t_{hi}\ \texttt{fresh} \qquad \otimes:\forall n.n\ bits \times n\ bits \to\ 2n\ bits \\ G_1 \overset{\mathscr{L}_1}{\subseteq} t_1 := e_1 \qquad G_2 \overset{\mathscr{L}_2}{\subseteq} t_2 := e_2 \qquad G_3 \overset{\mathscr{L}_3}{\subseteq}_M t_{hi},t_{lo} := \otimes(t_1,t_2) \\ G_4 \overset{\mathscr{L}}{\subseteq} \$m[loaddr(m,e,n)] := t_{lo} \mid \$m[hiaddr(m,e,n)] := t_{hi} \\ \mathscr{L}_1 \parallel uses(e_2) \cup uses(e) \qquad \mathscr{L}_2 \parallel uses(e) \qquad \mathscr{L}_3 \parallel uses(e) \end{array}}{G_1;G_2;G_3;G \overset{\mathscr{L}\cup\mathscr{L}_1\cup\mathscr{L}_2\cup L_3}{\subseteq} \$m[e] := \otimes(e_1,e_2)}$$

Figure 10. Tiling rules for standard and weird value operators

SXLOAD

$$\frac{\begin{array}{c} t'\ \texttt{fresh} \\ G \overset{\mathscr{L}}{\subseteq} t' := e \qquad G' \overset{\mathscr{L}'}{\subseteq}_M t := \texttt{sx}(\$m[t']) \qquad \mathscr{L} \parallel \{\$m[e]\} \end{array}}{G;G' \overset{\mathscr{L}\cup\mathscr{L}'}{\subseteq} t := \texttt{sx}(\$m[e])}$$

ZXLOAD

$$\frac{\begin{array}{c} t'\ \texttt{fresh} \\ G \overset{\mathscr{L}}{\subseteq} t' := e \qquad G' \overset{\mathscr{L}'}{\subseteq}_M t := \texttt{zx}(\$m[t']) \qquad \mathscr{L} \parallel \{\$m[e]\} \end{array}}{G;G' \overset{\mathscr{L}\cup\mathscr{L}'}{\subseteq} t := \texttt{zx}(\$m[e])}$$

LOSTORE

$$\frac{\begin{array}{c} t_1,t_2\ \texttt{fresh} \qquad \mathscr{L}_2 \parallel uses(e_1) \qquad G_1 \overset{\mathscr{L}_1}{\subseteq} t_1 := e_1 \\ G_2 \overset{\mathscr{L}_2}{\subseteq} t_2 := e_2 \qquad G \overset{\mathscr{L}}{\subseteq}_M \$m[t_1] := \texttt{lobits}_\texttt{n}(t_2) \end{array}}{G_2;G_1;G \overset{\mathscr{L}_1\cup\mathscr{L}_2\cup\mathscr{L}}{\subseteq} \$m[e_1] := \texttt{lobits}_\texttt{n}(e_2)}$$

Figure 11. Size-changing rules (registers $\leftrightarrow$ memory)

HWGETSX

$$\frac{G \overset{\mathscr{L}}{\subseteq}_M t := \texttt{sx}(r)}{G \overset{\mathscr{L}}{\subseteq} t := \texttt{sx}(r)}$$

HWGETZX

$$\frac{G \overset{\mathscr{L}}{\subseteq}_M t := \texttt{zx}(r)}{G \overset{\mathscr{L}}{\subseteq} t := \texttt{zx}(r)}$$

HWSET

$$\frac{t\ \texttt{fresh} \qquad G \overset{\mathscr{L}}{\subseteq} t := e \qquad G' \overset{\mathscr{L}'}{\subseteq}_M r := \texttt{lobits}_\texttt{n}(t)}{G;G' \overset{\mathscr{L}\cup\mathscr{L}'}{\subseteq} r := \texttt{lobits}_\texttt{n}(e)}$$

Figure 12. Size-changing rules (registers $\leftrightarrow$ small registers)

transfer words between registers and memory. Rule BLOCKCOPY copies data from one memory location to another, and it is not limited to a single word: a block copy can move any number of bytes (where a "byte" is the cell size of the memory space). Rule MOVEREG moves a word between registers; rule LI loads a compile-time constant into a register; and rule LI-LABEL loads a link-time constant into a register. Finally, rule MOVEEXP accounts for hardware restrictions on registers and operators. For example, if the result of an integer computation e is placed in a floating-point register t, the tiler introduces a fresh integer temporary t' to hold the result of the integer computation. Use of rule MOVEEXP reduces the burden on the implementor of the tileset: although the implementation must include a tile for every RTL operator (see, for example, rules BINOP and UNOP), it is sufficient that for each RTL operator, there is *some* set of registers in which the tiler can place the operands, and there is some set of registers in which the tiler can expect the result. It is not necessary to implement so many tiles that the result of *any* operator can be placed into *any* register. Our code-generation interface includes a mapping from each RTL operator into a data structure which identifies what kinds of registers or temporaries may be used as arguments and results for that operator.

6.5 Tiling control flow

To tile control flow (Figure 14), we use the final group of shapes in Figure 8. As with sequential control flow in Figure 9, the rules in Figure 14 express standard code-generation techniques. For example, if the target of an unconditional branch is known statically

$$\textsc{Store}\quad \frac{\begin{array}{c} t_1, t_2\ \texttt{fresh} \qquad \mathscr{L}_2 \parallel uses(e_1) \\ G_1 \overset{\mathscr{L}_1}{\subseteq} t_1 := e_1 \qquad G_2 \overset{\mathscr{L}_2}{\subseteq} t_2 := e_2 \qquad G \overset{\mathscr{L}}{\subseteq}_M \$m[t_1] := t_2 \end{array}}{G_2; G_1; G \overset{\mathscr{L}_1 \cup \mathscr{L}_2 \cup \mathscr{L}}{\subseteq} \$m[e_1] := e_2}$$

$$\textsc{Load}\quad \frac{\begin{array}{c} t'\ \texttt{fresh} \\ G \overset{\mathscr{L}}{\subseteq} t' := e \qquad G' \overset{\mathscr{L}'}{\subseteq}_M t := \$m[t'] \qquad \mathscr{L} \parallel \{\$m[e]\} \end{array}}{G; G' \overset{\mathscr{L} \cup \mathscr{L}'}{\subseteq} t := \$m[e]}$$

$$\textsc{BlockCopy}\quad \frac{\begin{array}{c} t_1, t_2\ \texttt{fresh} \qquad \mathscr{L}_1 \parallel \{\$m[e_2]\} \qquad \mathscr{L}_2 \parallel uses(e_1) \cup \{\$m[e_2]\} \\ G_1 \overset{\mathscr{L}_1}{\subseteq} t_1 := e_1 \qquad G_2 \overset{\mathscr{L}_2}{\subseteq} t_2 := e_2 \qquad G \overset{\mathscr{L}}{\subseteq}_M \$m[t_1] := \$m[t_2] \end{array}}{G_2; G_1; G \overset{\mathscr{L}_1 \cup \mathscr{L}_2 \cup \mathscr{L}}{\subseteq} \$m[e_1] := \$m[e_2]}$$

$$\textsc{MoveReg}\quad \frac{G \overset{\mathscr{L}}{\subseteq}_M t_1 := t_2}{G \overset{\mathscr{L}}{\subseteq} t_1 := t_2} \qquad \textsc{LI}\quad \frac{G \overset{\mathscr{L}}{\subseteq}_M t := k}{G \overset{\mathscr{L}}{\subseteq} t := k} \qquad \textsc{LI-Label}\quad \frac{G \overset{\mathscr{L}}{\subseteq}_M t := L}{G \overset{\mathscr{L}}{\subseteq} t := L}$$

$$\textsc{MoveExp}\quad \frac{t'\ \texttt{fresh} \qquad G \overset{\mathscr{L}}{\subseteq} t' := e \qquad G' \overset{\mathscr{L}'}{\subseteq}_M t := t'}{G; G' \overset{\mathscr{L} \cup \mathscr{L}'}{\subseteq} t := e}$$

Figure 13. Tiling rules for data movement

$$\textsc{BranchL}\quad \frac{G \overset{\mathscr{L}}{\subseteq}_M \mathbf{goto}\ L}{G \overset{\mathscr{L}}{\subseteq} \mathbf{goto}\ L} \qquad \textsc{BranchR}\quad \frac{\begin{array}{c} t\ \texttt{fresh} \qquad G \overset{\mathscr{L}}{\subseteq} t := e \\ G' \overset{\mathscr{L}'}{\subseteq}_M \mathbf{goto}\ t \end{array}}{G; G' \overset{\mathscr{L} \cup \mathscr{L}'}{\subseteq} \mathbf{goto}\ e}$$

$$\textsc{Compare}\quad \frac{\begin{array}{c} t_1, t_2\ \texttt{fresh} \\ ? : \forall n.n\ bits \times n\ bits \to \texttt{bool} \qquad G_1 \overset{\mathscr{L}_1}{\subseteq} t_1 := e_1 \\ G_2 \overset{\mathscr{L}_2}{\subseteq} t_2 := e_2 \qquad \mathscr{L}_1 \parallel uses(e_2) \qquad G \overset{\mathscr{L}}{\subseteq}_M L_T \triangleleft t_1 ? t_2 \triangleright L_F \end{array}}{G_1; G_2; G \overset{\mathscr{L}_1 \cup \mathscr{L}_2 \cup \mathscr{L}}{\subseteq} L_T \triangleleft e_1 ? e_2 \triangleright L_F}$$

Figure 14. Rules for tiling control flow

we use the **goto** L tile; otherwise we use the tiler to put the address into a fresh temporary t, then use the **goto** t tile, which typically corresponds to a "branch register" instruction.

The interesting rule in Figure 14 is the COMPARE rule: it applies only when the condition is a comparison. But in general, a condition can be any Boolean expression. If we look at the last block of Table 6 on page 580, we see that a Boolean expression can be produced not only by comparison operators, but also by Boolean constants or operators. The rules in Figure 15 show how to reduce any conditional branch to a control-flow graph in which every branch is either unconditional or is conditioned on the results of a comparison. These rules formalize standard techniques used when compiling Boolean expressions in a control-flow context. By considering every RTL operator whose return type is `bool`, we can be confident that these rules are complete.

$$\textsc{True}\quad \frac{G \overset{\mathscr{L}}{\subseteq}_M \mathbf{goto}\ L_T}{G \overset{\mathscr{L}}{\subseteq} L_T \triangleleft \mathbf{T} \triangleright L_F} \qquad \textsc{False}\quad \frac{G \overset{\mathscr{L}}{\subseteq}_M \mathbf{goto}\ L_F}{G \overset{\mathscr{L}}{\subseteq} L_T \triangleleft \mathbf{F} \triangleright L_F}$$

$$\textsc{Not}\quad \frac{G \overset{\mathscr{L}}{\subseteq} L_F \triangleleft e \triangleright L_T}{G \overset{\mathscr{L}}{\subseteq} L_T \triangleleft \texttt{not}(e) \triangleright L_F}$$

$$\textsc{Conjoin}\quad \frac{\begin{array}{c} L\ \texttt{fresh} \\ G \overset{\mathscr{L}}{\subseteq} L \triangleleft e_1 \triangleright L_F \qquad G' \overset{\mathscr{L}'}{\subseteq} L_T \triangleleft e_2 \triangleright L_F \qquad \mathscr{L} \parallel uses(e_2) \end{array}}{G; L\colon G' \overset{\mathscr{L} \cup \mathscr{L}'}{\subseteq} L_T \triangleleft \texttt{conjoin}(e_1, e_2) \triangleright L_F}$$

$$\textsc{Disjoin}\quad \frac{\begin{array}{c} L\ \texttt{fresh} \\ G \overset{\mathscr{L}}{\subseteq} L_T \triangleleft e_1 \triangleright L \qquad G' \overset{\mathscr{L}'}{\subseteq} L_T \triangleleft e_2 \triangleright L_F \qquad \mathscr{L} \parallel uses(e_2) \end{array}}{G; L\colon G' \overset{\mathscr{L} \cup \mathscr{L}'}{\subseteq} L_T \triangleleft \texttt{disjoin}(e_1, e_2) \triangleright L_F}$$

Figure 15. Rules to eliminate Boolean operators

We have formulated the COMPARE rule in a way which glosses over the fact that a conditional-branch *node* in a control-flow graph must be reduced to a *sequence* of machine instructions. The COMPARE rule shows an *unserialized* control-flow graph, in which the true and false labels L_T and L_F are treated symmetrically. But the `bc` shape in Figure 8 does not treat the labels symmetrically; when the condition is not true, the shape falls through to the successor instruction. The judgment $G \subseteq^{\mathscr{L}}_M L_T \triangleleft t_1 ? t_2 \triangleright L_F$ actually provides *two* tiles, one of which is chosen when the flow graph is serialized. The graph may be serialized either as $L_T \triangleleft t_1 ? t_2 \triangleright L'; L'\colon \mathbf{goto}\ L_F$, where L' is fresh, or as $L_F \triangleleft \neg(t_1 ? t_2) \triangleright L'; L'\colon \mathbf{goto}\ L_T$, where L' is again fresh. Each of these serializations uses a tile of the `bc` shape and a tile of the `b` (unconditional branch) shape. The basic blocks of a control-flow graph are serialized using a reverse postorder depth-first traversal, which in most cases allows the serializer to drop the fresh label L' and the unconditional branch.

6.6 Informal argument about completeness

Our tiler addresses a different problem from the instruction selectors found in most compilers. Most code-generation problems start with a known source language and an intermediate code tailored to that language, and all programs in the source language must be compiled. But C-- is not a source language: it is a low-level target language for compilers. It resembles low-level intermediate codes used in compilers, but it is not tailored to any particular source language; rather, it is tailored to express today's architectural consensus on floating-point computation, integer computation, and bitwise computation—on words of any width. Our code-generation problem starts with an *unknown* source language, which compiles to a *subset* of C--, which we must tile to machine instructions.

We do not expect to compile all well-typed C-- programs. For example, a program that uses 17-bit bytes and 73-bit arithmetic may be well typed, but we cannot compile it. We compile only code that respects the *memory-cell size*, *byte order*, and *word size* of the target machine. For example, code intended to run on the SPARC should use 8-bit cells (bytes) in memory, big-endian loads and stores, and 32-bit RTL operators. Code intended to run on a PDP-10 should use 36-bit cells (words) in memory, 36-bit loads and stores, and 36-bit RTL operators.

With integer and bitwise operations, we are more flexible: a compiler pass called the *widener* converts narrow operations to operations at the word size of the target machine. For example, the widener can translate 32-bit code to 64-bit code while minimizing sign extensions and zero extensions (Redwine and Ramsey 2004).

If a control-flow graph G contains only well-typed RTLs which use the proper memory size, byte order, and word size, and which have trivial guards, our tiler reduces G to tiles, and thence to machine instructions. We argue by structural induction over the graph. Section 6.1 shows how sequences and parallel assignments are reduced to individual graphs and single assignments. Section 6.5 shows how a control-flow graph containing a nontrivial guard is reduced to conditional-branch tiles. The most complex part of the induction is over assignments. Given only value operators and data movement, our rules cover all the type schemes of all the `C--` operators. A new RTL operator can easily be added without changing the tiler, provided the operator has the same type as an existing RTL operator. In all cases, the side conditions on $\mathscr{L}$ can be satisfied by choosing a combination of fresh temporaries and scratch registers.

The difficult part of the argument lies with size-changing operators. We believe that our tiler is complete for `C--` programs in which values are sign-extended or zero-extended to at most the word size, and low bits are extracted from words, not from larger values. Unfortunately, these restrictions rule out cases of practical interest. For example, on the SPARC, it can be necessary to convert a double-precision floating-point number to a 64-bit integer, then transfer the most and least significant words of that integer to two 32-bit integer registers. This case is supported by the Quick `C--` compiler, but we do not have a general algorithm for translating size-changing operators when operands or results are larger than one word, so the tiling rules for this case are omitted from this paper.

7. Discussion

Our main result is a machine-independent tileset and tiling algorithm that can be used with any register machine. The tileset and algorithm are implemented in our Quick `C--` compiler, as are a similar tileset and algorithm for stack machines. Experimental results are good, although they reflect the immaturity of Quick `C--`'s optimizer: our generated code outperforms code generated by `lcc` or by gcc with optimization turned off, but it is not as good as code generated by gcc with optimization turned on. Details of these results have already been published (Dias and Ramsey 2010). Our results, together with Benitez and Davidson's (1994) demonstration that standard scalar and loop optimizations can be implemented in a compiler very similar to Quick `C--`, indicate that our tiler could be used in a high-quality optimizing compiler.

Our approach to instruction selection is built around two ideas:

- Instead of designing a new tileset for every target machine, design a single, machine-independent tileset that is reused for every machine in a big architectural family.
- Let the design of the tileset follow from a formal model of machine-level computation (RTLs), which expresses the consensus common to the architectural family.

These ideas make it possible to reduce the intellectual work required to add support for new target machines.

The consensus common to the architectural family of register machines includes flat memory addressing, a preferred word size and byte order, and groups of interchangeable registers; these features are expected to be found in all possible targets, and in those aspects our design is a classic *intersection machine*. But our design also supports a rich and extensible set of computational operators, and in that aspect our design is a classic *union machine*.

We establish the correctness and completeness of the tiling algorithm just once per architectural family, instead of once per target machine. While arguing about completeness can be difficult, the work we do per machine—to show that our tileset is implemented correctly and completely—is trivially easy. The reduction in overall work represents a significant advance over prior art. To illustrate the work, we conclude by discussing how you might create an infrastructure, add new languages, add new target machines, and add new optimizations, and how our techniques (together with those of Davidson and Fraser) will help you separate concerns.

To create an infrastructure, you will define representations of RTLs and control-flow graphs, and you will define an interface that allows a front end to create control-flow graphs. You will specify a code-generation interface for our tileset. If you wish to generate code for the x86 legacy floating-point unit or for some other stack machine, you will also specify an interface for a stack-machine tileset, perhaps like the one in Appendix A. You will implement the tiler. And if you choose to generate machine-dependent components from declarative machine descriptions (Ramsey and Davidson 1998), you will implement a recognizer generator and a tileset generator that understand your representation of RTLs (Dias and Ramsey 2006, 2010).

To add a new language, you will build a front end that translates your language to low-level, imperative intermediate code. For example, if your language includes first-class, nested functions, you will write a front end that performs closure conversion, defunctionalization, or some other translation to first-order code. Your primary concern will be the semantics of your language, but you will also need a thorough grasp of machine-level computation, and your front end will need to generate code that matches the word size and byte order of the target machine. You will be insulated from all other details of the target machine's instruction set.

To add a new target machine, you will need to implement a recognizer and our tileset for that machine. You can do this by hand, or you can generate a recognizer and an implementation of the tileset from a declarative machine description. Your primary concern will be to know what target-machine instructions are useful and what their semantics are. You will be insulated from details of optimization and from the front end. If your infrastructure generates the recognizer and the implementation of our tileset, you will also be insulated from the details of your compiler's intermediate representation and from the details of writing the recognizer and implementing the tileset—and you may be able to reuse a machine description written by someone else.

To add an optimization, you will write a code-improving transformation on RTLs. Most likely your primary concern will be to implement a machine-independent code improvement—ideally one that can improve almost any RTL and so will be effective on many target machines. But you may instead be concerned with implementing a machine-independent transformation which is intended to be effective only on particular machines, for example, a vectorizing transformation. Either way, you will be insulated from irrelevant details of the target-machine architecture: the recognizer will ensure that your optimization preserves the machine invariant and is safe to run on any machine.

This plan of building an infrastructure and then adding to it incrementally, which we have done with Quick `C--`, does not require radical departures from established ways of doing things, and it is not limited to a particular source language, intermediate representation, or target machine. Indeed, our design can be thought of as a modest refactoring of a Davidson/Fraser compiler. And yet, as with many good ideas in software design, modest changes yield significant benefits: our design separates front end from back end and IR from target machine to a degree not achieved in previous compilers. When combined with automatic generation of machine-dependent components, our plan offers a uniquely cost-effective way of building quality compilers.

Acknowledgments

Kevin Redwine carefully scrutinized the tiling rules. Eddie Aftandilian helped improve the presentation. The anonymous referees provided invaluable feedback about the presentation and the work. In particular, referee A's review showed extraordinary depth, insight, and attention to detail.

The work was funded by a grant from Intel Corporation and by NSF awards 0838899 and 0311482.

References

Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. 1989 (October). Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11 (4):491–516.

Andrew W. Appel. 1998. *Modern Compiler Implementation*. Cambridge University Press, Cambridge, UK. Available in three editions: C, Java, and ML.

Manuel E. Benitez and Jack W. Davidson. 1988 (July). A portable global optimizer and linker. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 23(7):329–338.

Manuel E. Benitez and Jack W. Davidson. 1994 (March). The advantages of machine-dependent global optimization. In *Programming Languages and System Architectures*, *LNCS* volume 782, pages 105–124. Springer Verlag.

Roderic G. G. Cattell. 1980 (April). Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems*, 2(2):173–190.

Melvin E. Conway. 1958 (October). Proposal for an UNCOL. *Communications of the ACM*, 1(10):5–8.

Jack W. Davidson and Christopher W. Fraser. 1984 (October). Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526.

João Dias and Norman Ramsey. 2006 (March). Converting intermediate code to assembly code using declarative machine descriptions. In *15th International Conference on Compiler Construction (CC 2006)*, *LNCS* volume 3923, pages 217–231.

João Dias and Norman Ramsey. 2010 (January). Automatically generating back ends using declarative machine descriptions. In *Proceedings of the 37th ACM Symposium on the Principles of Programming Languages*, pages 403–416.

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

Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. 1992 (September). Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226.

Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. 1992 (April). BURG—fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76.

Jean-Yves Girard. 1986. The System F of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86.

Tim Lindholm and Frank Yellin. 1999. *Java Virtual Machine Specification*. Addison-Wesley, second edition.

Robin Milner. 1978 (December). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

Eduardo Pelegrí-Llopart and Susan L. Graham. 1988 (January). Optimal code generation for expression trees: An application of BURS theory. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 294–308.

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

Todd A. Proebsting. 1992 (June). Simple and efficient BURS table generation. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 27 (7):331–340.

Norman Ramsey and Cristina Cifuentes. 2003 (March). A transformational approach to binary translation of delayed branches. *ACM Transactions on Programming Languages and Systems*, 25(2):210–224.

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

Norman Ramsey, Simon Peyton Jones, and Christian Lindig. 2005 (February). The C`--` language specification Version 2.0 (CVS revision 1.128). See `http://www.cminusminus.org/code.html#spec`.

Norman Ramsey and Simon L. Peyton Jones. 2000 (May). 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.

Kevin Redwine and Norman Ramsey. 2004 (April). Widening integer arithmetic. In *13th International Conference on Compiler Construction (CC 2004)*, *LNCS* volume 2985, pages 232–249.

J. Strong, J. H. Wegstein, A. Tritter, J. Olsztyn, Owen R. Mock, and T. Steel. 1958. The problem of programming communication with changing machines: A proposed solution (Part 2). *Communications of the ACM*, 1 (9):9–16.

Henry S. Warren. 2003. *Hacker's Delight*. Addison-Wesley.

A. Shapes of tiles for a stack machine

These shapes are used in our tiler for the x86 legacy floating-point unit. The stack pointer is `st`, and the stack grows downward.

Standard and weird value operators

$f[\mathtt{st}] := \oplus(f[\mathtt{st}])$	`stackop`
$f[\mathtt{st}+1] := \oplus(f[\mathtt{st}], f[\mathtt{st}+1]) \mid \mathtt{st} := \mathtt{st}+1$	`stackop`
$f[\mathtt{st}] := \oplus(f[\mathtt{st}], rm)$	`stackop_rm`
$f[\mathtt{st}+1] := \oplus(f[\mathtt{st}], f[\mathtt{st}+1], rm) \mid \mathtt{st} := \mathtt{st}+1$	`stackop_rm`

Size-changing operators

$m[t_1] := \oplus(f[\mathtt{st}]) \mid \mathtt{st} := \mathtt{st}+1$	`store_pop_cvt`
$f[\mathtt{st}-1] := \oplus(m[t_1]) \mid \mathtt{st} := \mathtt{st}-1$	`push_cvt`
$m[t_1] := \oplus(f[\mathtt{st}], rm) \mid \mathtt{st} := \mathtt{st}+1$	`store_pop_cvt_rm`
$f[\mathtt{st}-1] := \oplus(m[t_1], rm) \mid \mathtt{st} := \mathtt{st}-1$	`push_cvt_rm`

Data movement (most tiles transfer exactly one word)

$m[t_1] := f[\mathtt{st}] \mid \mathtt{st} := \mathtt{st}+1$	`store_pop`
$f[\mathtt{st}-1] := m[t_1] \mid \mathtt{st} := \mathtt{st}-1$	`push`
$f[\mathtt{st}-1] := k \mid \mathtt{st} := \mathtt{st}-1$	`pushk`
$f[\mathtt{st}-1] := \oplus(k) \mid \mathtt{st} := \mathtt{st}-1$	`pushk_cvt`

Control flow (including comparison operators)

$L_T \triangleleft f[\mathtt{st}]?\, f[\mathtt{st}+1] \triangleright L_F; L_F$:	`bc_stack`