Reprinted from *Proceedings of the 2010 ACM Symposium on Principles of Programming Languages* (POPL'10)

Automatically Generating Instruction Selectors Using Declarative Machine Descriptions

João Dias
Tufts University
dias@cs.tufts.edu

Norman Ramsey
Tufts University
nr@cs.tufts.edu

Abstract

Despite years of work on retargetable compilers, creating a good, reliable back end for an optimizing compiler still entails a lot of hard work. Moreover, a critical component of the back end—the instruction selector—must be written by a person who is expert in both the compiler's intermediate code and the target machine's instruction set. By *generating* the instruction selector from declarative machine descriptions we have (a) made it unnecessary for one person to be both a compiler expert and a machine expert, and (b) made creating an optimizing back end easier than ever before.

Our achievement rests on two new results. First, finding a mapping from intermediate code to machine code is an undecidable problem. Second, using heuristic search, we can find mappings for machines of practical interest in at most a few minutes of CPU time.

Our most significant new idea is that heuristic search should be controlled by algebraic laws. Laws are used not only to show when a sequence of instructions implements part of an intermediate code, but also to limit the search: we drop a sequence of instructions not when it gets too long or when it computes too complicated a result, but when *too much reasoning* will be required to show that the result computed might be useful.

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

General Terms Algorithms, Experimentation, Theory

Keywords Instruction selection, retargetable compilers, declarative machine descriptions

1. Introduction

Writing a back end for an optimizing compiler is difficult. A compiler writer must know not only the syntax and semantics of a target instruction set but also the internal data structures and invariants of a compiler, which can be very complex. For years, the state of the art has been to write instruction selectors using domain-specific languages which combine knowledge of a compiler's data structures with knowledge of a target machine (Aho, Ganapathi, and Tjiang 1989; Emmelmann, Schröer, and Landwehr 1989; Fraser 1989; Fraser, Henry, and Proebsting 1992; Fraser, Hanson, and Proebsting 1992). We present a new technique which realizes a long-sought goal: decoupling compiler knowledge from machine knowledge (Conway 1958; Strong et al. 1958). Using our technique, an instruction selector is generated automatically from *declarative machine descriptions*. (A declarative machine description contains no code and no information about any compiler's data structures; instead, it simply and formally describes properties of a target machine.)

Our contributions are as follows:

- We show that given a description of an arbitrary instruction set, generating an instruction selector is undecidable (Section 8). To find machine instructions that implement intermediate code, it is therefore necessary to search heuristically.
- We present a new heuristic-search algorithm, which starts with the expressions computed by the machine's instruction set and gradually adds to a pool of computable expressions until every intermediate-code expression is computable.

  A crucial invariant is that we consider *only* computations that we *know* can be implemented entirely by machine instructions. This invariant makes our algorithm significantly simpler than earlier search algorithms, which start with goal computations whose implementations by machine instructions are not known.
- To increase the pool of computable expressions, we rewrite existing computable expressions using algebraic laws. To match the left-hand side of an algebraic law, we have developed a new algorithm called *establishment*, which uses a novel combination of unification and machine code to make two expressions equal (Section 7, especially Figure 4).
- Finally, we present a new pruning mechanism which does not restrict the size of a computable expression or the length of the sequence of machine instructions used to compute an expression. Instead, we restrict the *amount of reasoning* that is expected to be applied to reach a goal (Section 9).

Our heuristic search easily finds instruction sequences for popular hardware. We have generated working back ends for *x*86, PowerPC, and ARM targets, which are representative of desktops, game consoles, and handhelds. By selecting instructions *before* optimizing, our back ends generate code that is as good as code generated by hand-written back ends (Section 10.1). As a result of our work, a good, reliable back end for an optimizing compiler can be built more quickly and easily than ever before.

2. Instruction selection and register transfers

We use a design developed by Davidson and Fraser (1980; 1984): a *code expander* translates intermediate code to low-level *register-transfer lists* (RTLs). The code expander establishes the *machine invariant*: each RTL is implementable by a single instruction on the target machine. An optimizer improves the RTLs while preserving the machine invariant. Finally, the compiler emits assembly code by translating each RTL to a corresponding machine instruction.

POPL'10, January 17–23, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-479-9/10/01. . . $10.00
Reprinted from POPL'10,, [Unknown Proceedings], January 17–23, 2010, Madrid, Spain., pp. 403–416.

Literal constant	k		
RTL operator	$\oplus$		
Storage space	s		
Location set	ls		
Expression	e, p	$::=$	$x_{\#w} \mid k \mid l \mid \oplus(e_1, \ldots, e_n)$
Location	l	$::=$	$x_{\in ls} \mid \$s[e]$
Assignment	A	$::=$	$l := e$
Guarded assignment	g	$::=$	$p \rightarrow A$
RTL	R	$::=$	$g \; \{ \mid g \}$
Control-flow graph	G	$::=$	$\epsilon \mid R; G$

Figure 1. Grammar for register transfers.

To make this approach practical, the optimizer's code-improving transformations must be independent of the machine invariant. Davidson and Fraser's (1980) idea is to have the optimizer call a machine-dependent *recognizer*, which enforces the machine invariant. A code improvement is accepted only if the recognizer says that each RTL in the improved code is implementable by a single instruction on the target machine. We have generated recognizers from declarative machine descriptions (Dias and Ramsey 2006).

Even though the optimizer works at a very low level, Davidson and Fraser's approach can easily implement all the scalar and loop optimizations expected of an optimizing compiler (Benitez and Davidson 1994). The approach is ideal for our work because an automatically generated code expander need only generate *correct* code; *good* code is not required. In fact, Davidson (2008) reports that this approach works best when starting with naïve code.

Before explaining how to *generate* a code expander (Section 6), we cover background: how RTLs represent instructions (Section 3), what is extracted from a declarative description of the semantics of instructions (Section 4), and how our code expander is divided into machine-independent and machine-dependent parts (Section 5).

3. Representing instructions as RTLs

Register transfers are composed primarily from *expressions*, *locations*, *assignments*, and *RTL operators* (Figure 1). When analyzing an instruction set at compile-compile time, we also allow *operand metavariables* $x_{\#w}$ and $x_{\in ls}$ to stand for expressions and locations, respectively. Internally, each expression, location, and operator has a width, but these widths are typically inferred by a type checker, and for simplicity, in this paper we usually leave widths implicit.

At compile time, an expression e is a literal constant k, a fetch from a location l, or the application of an RTL operator $\oplus$ to subexpressions. An RTL operator represents a pure computation on bit vectors; our library of about 90 operators includes standard integer, logical, and IEEE floating-point operations. If these don't suffice to describe the semantics of an instruction, a machine description can introduce a new operator. Operator applications are written using prefix notation, but for binary operators we use infix notation too.

At compile time, a location l is one or more contiguous *cells* in a *storage space* s. Storage spaces include memory, general-purpose registers, and special-purpose registers; collectively they form the machine state. Each storage space gets a one-character name; we conventionally use `r` for general-purpose registers, `f` for floating-point registers, and `m` for memory. Locations are referred to using an array-index notation, but for convenience, we also name locations; for example, on the *x*86, the name `ESP` refers to location `$r[4]`: general-purpose register 4, the stack pointer.

An assignment computes the value of an expression and stores the result in a location. A guarded assignment $p \rightarrow l := e$ evaluates a predicate p called the *guard*, and if p is true, e is evaluated and the result is stored in l. For ordinary instructions, the guard is trivially true and is omitted; guarded assignments with nontrivial guards typically represent conditional-branches or predicated instructions.

A register-transfer list or RTL R is a parallel composition of guarded assignments; it is like Dijkstra's (1976) multiple assignment except that each assignment has its own guard. A single RTL can represent the input/output behavior of any machine instruction.

Our algorithm, like our compiler, works with control-flow graphs G. For simplicity, in this paper we assume that a control-flow graph is a sequence of RTLs; an empty sequence is written ϵ.

4. Declarative machine descriptions and λ-RTL

A declarative machine description specifies properties of a machine using formalism that is independent of any particular tool or programming language. Properties of interest for code generation include the semantics and the binary or assembly-language representations of machine instructions. By design, a declarative machine description can be written by a machine expert who knows nothing about the internals of any compiler.

A declarative machine description can be used to generate parts of many tools, not just a compiler back end. For example, the declarative semantic-description language λ-RTL (Ramsey and Davidson 1998) and its sister language SLED (Ramsey and Fernández 1997) have been used to help generate assemblers and disassemblers (Ramsey and Fernández 1995), linkers (Fernández 1995), dynamic code generators (Auslander et al. 1996), debuggers, and binary translators (Cifuentes, Van Emmerik, and Ramsey 1999).

A declarative machine description is not a program in a domain-specific language: extracting a program requires analysis. The contribution of this paper is a set of analyses that together solve the most difficult and important problem in this field: extracting a translation of intermediate code into machine instructions.

λ-RTL and SLED model an instruction set using an algebraic datatype that represents the abstract syntax of instructions. Each instruction corresponds to a *constructor*, which can be applied to operands. For example, the PowerPC's three-register add instruction has the abstract syntax `add(rd, rs1, rs2)`; the `add` constructor is applied to three operands. An operand may be a bit vector or the result of applying another constructor.

λ-RTL's algebraic datatypes are equivalent to a grammar with named productions. This equivalence makes it easy to explain how λ-RTL specifies semantics: a λ-RTL description defines an *attribute grammar* in which the meaning of an instruction is a synthesized attribute in the form of an RTL with metavariables. Each production in the grammar is associated with an equation that tells how to synthesize a result attribute from the attributes of its operands. For example, the PowerPC `add` instruction is specified as follows:

```
default attribute
  add(rd, rs1, rs2) is $r[rd] := $r[rs1] + $r[rs2]
```

The `rd`, `rs1`, and `rs2` on the left-hand side are binding instances. These metavariables have been declared as 5-bit operands; in the language of attribute grammars, they are "terminal symbols." When an RTL is created at compile time, metavariables are instantiated with values, so for example the abstract-syntax tree `add(1, 2, 3)` has the semantics `$r[1] := $r[2] + $r[3]`.

RTL metavariables have the forms $x_{\#w}$ and $x_{\in ls}$. A metavariable of the form $x_{\#w}$ ranges over bit vectors of width w. A metavariable of the form $x_{\in ls}$ ranges over locations in set ls. In our `add` example, a metavariable $x_{\#5}$ could stand for the register number `rs1`; a metavariable $x_{\in \texttt{\$r[0..31]}}$ could stand for the location `$r[rs1]`, which might be any register in space `r`.

From a λ-RTL description, we extract the following information:

- A grammar from which all instructions can be generated
- A list of storage spaces and locations
- Equations that show how to compute the observable effect of any instruction, represented as an RTL with metavariables

5. Code expansion by tiling

A code expander translates a statement in an input language into a control-flow graph G in which each graph node holds an RTL that is implementable by a single instruction on the target machine. Our input language is C--, whose ASCII syntax is a thin veneer over RTLs (Ramsey and Peyton Jones 2000). C-- is easily targeted by a front end: a front end may emit any assignment so long as no value or intermediate result is wider than a machine word. The front end, therefore, needs to know almost nothing about the target machine: only the word size and possibly the byte order.

Because a front end can generate almost any RTL, a code expander for C-- has to work harder than a compiler for a language like C or Java: it must translate any well-formed RTL to equivalent RTLs that satisfy the machine invariant. To make it easier to write code expanders, we factor each expander into two parts: a set of *expansion tiles* (or *tileset*) provides machine-dependent implementations for a fixed, machine-*independent* set of RTLs; and a machine-independent *tiler* translates any well-formed RTL into an equivalent control-flow graph whose nodes hold tiles from the tileset.

Our tiler is implemented by tree covering (Pelegrí-Llopart and Graham 1988; Aho, Ganapathi, and Tjiang 1989; Emmelmann, Schröer, and Landwehr 1989; Fraser, Hanson, and Proebsting 1992). What distinguishes our tiler from previous work is that no matter what the target machine, *every back end provides the same set of tiles and uses the same tiler* (Dias 2008, Chapter 5). The tiler uses maximal munch to translate a well-typed RTL to a sequence of tiles. For example, given the memory-to-memory move

$$\$m[\texttt{ESP}+8] := \$m[\texttt{ESP}+12]$$

the tiler may generate a sequence of six tiles, each of which either moves a value or applies a single RTL operator:

$$\begin{aligned}
t_1 &:= 12 && \text{// load-immediate tile}\\
t_2 &:= \texttt{ESP} + t_1 && \text{// binop tile}\\
t_3 &:= \$m[t_2] && \text{// load tile}\\
t_4 &:= 8 && \text{// load-immediate tile}\\
t_5 &:= \texttt{ESP} + t_4 && \text{// binop tile}\\
\$m[t_5] &:= t_3 && \text{// store tile}
\end{aligned}$$

The example shows why this transformation is called *code expansion*: a single RTL is expanded into many tiles. The expansion factor is large because the tiles are small: each tile implements at most one RTL operator. Each tile in turn will be implemented by a sequence of machine instructions, but because a tile can usually be implemented using one or two instructions, the remaining expansion factor is smaller. Even so, the expanded code can be horribly inefficient—but it is ideal for peephole optimization and other optimizations that take place after instruction selection (Davidson and Fraser 1984; Benitez and Davidson 1994).

Our machine-independent tileset is designed to cover all well-typed RTLs (Dias 2008, Chapter 5). We have defined dozens of tiles: one for each RTL operator, plus tiles for data movement and control transfer. But what matters to our tiler is not the number of tiles but the number of forms, which we call *shapes*. The shape of a tile is obtained by abstracting over RTL operators and over the widths of arguments and results, and the size of the tiler is proportional to the number of shapes, not the number of tiles.

Because the tiler is machine-independent, *we have reduced the problem of generating an instruction selector to the problem of finding an implementation of each tile using only target-machine instructions*. Before moving on to this problem, we discuss the structure of the tileset:

- An *operator tile* applies a *single* RTL operator to constant or register operands and places a result in one or more registers. Operator tiles come in 7 shapes.
- A *data-movement tile* moves a constant into a location or moves a value between two locations. There are shapes for register-register copies, loads from memory, and stores to memory. Data-movement tiles also move data between locations of different widths, so there is a shape for sign-extending and zero-extending loads and a shape for instructions that store the low bits of a register in a memory location. In total, data-movement tiles come in 13 shapes.
- A *control-transfer tile* changes the flow of control. There are shapes for conditional and unconditional branches, indirect branches, direct and indirect calls, and returns. Control-transfer tiles come in 9 shapes.

The shapes above are for register machines. Stack machines, such as the *x*86 legacy floating-point unit, add another dozen shapes.

6. Our algorithm for finding tiles

Finding *implementations* of tiles turns out to be undecidable (Section 8). We have therefore developed a new heuristic search algorithm, which works well for machines of practical interest.

Our algorithm resembles answer-set programming. We maintain a *pool* of RTLs, each of which is known to be implementable by a sequence of instructions on the target machine. The pool is initialized with RTLs from the declarative description of the target machine. These RTLs may or may not include some tiles. To find more tiles, we *combine RTLs in the pool*, creating new RTLs which are also implementable by machine instructions. We continue adding RTLs to the pool until we have found an implementation of each tile or until our pruning heuristic tells us to stop. In the rest of this section, we refine the description of our algorithm, with examples. Formal development is deferred to Section 7.

6.1 Intuitions about the algorithm

We grow a pool of RTLs that we know how to implement on the target machine. We therefore maintain the following invariant: each RTL in the pool has an implementation that uses only instructions which exist on the target machine. We begin by populating the pool with RTLs from the λ-RTL description:

- Although we talk about RTLs, an element of the pool is actually a control-flow graph G, together with a postcondition that says what expression e is computed by G. Both G and e are parameterized by operand metavariables $x_{\#w}$ and $x_{\in ls}$. Initially, every graph G is a machine instruction, and the associated postcondition gives the contents of one location assigned to by G.

We enlarge the pool using the following ideas:

- *Sequence RTLs to implement new RTLs*: For example, if the pool contains RTLs $l_1 := e_1$ and $l_2 := e_2$ such that e_2 uses location l_1, we can combine the RTLs in sequence so the result stored into l_2 will be $e_2[l_1 \mapsto e_1]$, where e_1 is substituted for l_1.
- *Compensate for unwanted assignments*: A tile assigns to just one location. But a sequence of RTLs may assign to many locations. To use such a sequence, we compensate for unwanted assignments by making sure they cannot be observed.

- *Apply algebraic laws*: An algebraic law asserts that two expressions are equivalent. If a sequence of RTLs computes a complicated expression, we might be able to use an algebraic law to show that the same sequence computes a simpler expression—like the right-hand side of an expansion tile.

If by using these ideas, we get a sequence of instructions that computes an expression e which is not already computed by some other RTL in the pool, then provided e satisfies the pruning heuristic described in Section 9, we add the sequence to the pool. Our pruning heuristic predicts whether an assignment is likely to help implement a tile. Pruning not only ensures termination but also reduces the time spent finding new combinations of RTLs.

We illustrate our algorithm with an example: finding an implementation of the bitwise-complement tile $r_1 := \neg r_2$ on the PowerPC. The initial pool of RTLs does not include bitwise complement, but it does include a load-zero instruction[1] and a bitwise nor:

$$G_{ldzero} \triangleq r_3 := 0$$
$$G_{bitnor} \triangleq r_1 := \neg(r_2 \vee r_3)$$

Here r_1, r_2, and r_3 are RTL metavariables standing for general-purpose registers. We can combine these instructions in a sequence

$$G_{com+} \triangleq r_3 := 0; r_1 := \neg(r_2 \vee r_3)$$

which computes both the bitwise nor and the assignment to r_3:

$$G_{com+} \equiv r_1 := \neg(r_2 \vee 0) \mid r_3 := 0$$

Now we can apply the algebraic law $x \vee 0 = x$, with r_2 substituted for x, to simplify the bitwise-nor expression:

$$G_{com+} \equiv r_1 := \neg r_2 \mid r_3 := 0$$

The RTL $r_1 := \neg r_2 \mid r_3 := 0$ computes bitwise complement, but it also assigns zero to r_3. To compensate for the assignment to r_3, we have to make it unobservable by the rest of the program. One way is to replace r_3 with a location that cannot be observed: a fresh temporary or a scratch register.

A fresh temporary is guaranteed to be distinct from every other location in the program; for most sets of machine registers, the compiler provides an infinite supply of fresh temporaries. Temporaries are eventually mapped to hardware registers by a register allocator. If we substitute a fresh temporary t for the metavariable r_3, the assignment to t cannot be observed:

$$G_{com} \triangleq G_{com+}[r_3 \mapsto t] \equiv r_1 := \neg r_2 \mid t := 0$$

Not every hardware register has associated temporaries. For example, a unique hardware resource like a condition-code or status register would not have temporaries. In this case, the best option is for the back end to designate the resource as a *scratch register*. A scratch register is made unnameable in source code and unavailable to the register allocator. Scratch registers may be mutated freely in the implementation of any tile.

Finally, if we have an unwanted assignment to a location that has no temporaries and is not a scratch register, our only choice is to insert code to save and restore the value in the location. This option is workable only in straight-line code; to save before and restore after a branch instruction would require inserting code at the branch target, which would violate the abstraction of the tileset.

In our example graph G_{com}, the only observable assignment is $r_1 := \neg r_2$, which implements the bitwise-complement tile, and we add G_{com} to the pool.

6.2 Algebraic Laws

An algebraic law defines an axiom used to show that two RTL expressions are equivalent. The language of algebraic laws uses the same operators as the language of RTL expressions, but it refers only to constants and to special algebraic-law metavariables x_L, never to locations:

Law metavariable	x_L
Law expression	$e^{\text{LAW}} ::= x_L \mid k \mid \oplus(e_1^{\text{LAW}}, \ldots, e_n^{\text{LAW}})$
Law	$\mathit{Law} ::= e_1^{\text{LAW}} = e_2^{\text{LAW}}$

When we apply an algebraic law, its metavariables may be replaced by *any* RTL expressions e, not only expressions of the form e^{LAW}. To improve readability, we usually omit the subscript L of an algebraic-law metavariable.

Algebraic laws are universally true, independent of any machine. The most familiar kinds of laws are identities and inverses, such as $x + 0 = x$ and $\neg\neg x = x$. Another familiar kind of law shows how to implement one RTL operator in terms of other RTL operators; for example, $\neg x = -x - 1$. A more interesting kind of law shows how a value can be equal to an expression involving that same value; for example, the following law is used to find instruction sequences that load large immediate values on RISC architectures:

$$((x \gg_l n) \ll n) \vee \texttt{zx}(\texttt{lobits}_n(x)) = x$$

where zx is zero-extension and $\texttt{lobits}_n$ is an operator that extracts the low n bits of its argument. There are many such laws, which embody both machine knowledge and some of the kind of domain-specific knowledge presented by Warren (2003).

6.3 Data movement

To save and restore mutated locations and to implement data-movement tiles, we must find sequences of instructions that move data between locations. We put such sequences in a directed *data-movement graph*. Each node in the graph represents a set of locations on the machine, determined using Feigenbaum's (2001) *location-set analysis*, as adapted by Dias and Ramsey (2006). This analysis identifies locations that are interchangeable for use in some instructions. For example, most machines have general-purpose registers that are interchangeable in most instructions. (Particular registers may have special, noninterchangeable status in a few instructions such as multiply or divide.) For each RTL in the pool that moves data from location l' to location l, we add the edge $l' \rightarrow l$ to the data-movement graph. We then compute the transitive closure of the graph, so if a sequence of RTLs G moves data from l_1 to l_2, we add another edge to the data-movement graph, and we associate G with that edge. For example, to show that we can move data between integer registers on the *x*86, we add an edge from the location set `$r[0..7]` to itself; we represent the edge using two metavariables ($x_{\in\texttt{\$r[0..7]}} \rightarrow x'_{\in\texttt{\$r[0..7]}}$); and we associate the edge with the RTL for the move instruction, with the metavariables $x_{\in\texttt{\$r[0..7]}}$ and $x'_{\in\texttt{\$r[0..7]}}$ as operands.

An ideal data-movement graph would be complete: able to move a value from any location on the machine to any other location of the same size. Real data-movement graphs usually lack some edges that involve special registers, such as condition-code registers.

7. Formal development

The essence of our algorithm is to find a control-flow graph G, composed of machine instructions, such that when G is executed, it implements an assignment of the form $l := e$, which is the form of almost every tile. When we discover a graph that implements a new assignment, what we add to the pool is not an RTL but rather

[1] Actually the PowerPC instruction set does not include a load-zero instruction, but the load-zero simplifies this example. The graph G_{ldzero} is derived by our algorithm; it is equivalent to the graph labelled D in Figure 6.

CREATE CANDIDATE

$$\frac{\begin{array}{c} e_1^{\mathrm{LAW}} = e_2^{\mathrm{LAW}} \in \mathit{Laws} \\ \cdot \vdash \{e = e_1^{\mathrm{LAW}}\}\ G_0\ \{l_0 = \mathbf{in}\ E[e_2^{\mathrm{LAW}}] \wedge \mathit{modifies}\ \mathcal{L}_0\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{l' = \mathbf{in}\ e' \wedge \mathit{modifies}\ \mathcal{L}'\} \\ \langle G_s, G_r, \mathcal{L}_s, \theta_{sr}\rangle \Updownarrow \mathrm{observable}(\mathcal{L}') \end{array}}{\{\mathit{true}\}\ G_0\ \{l_0 = \mathbf{in}\ E[e] \wedge \mathit{modifies}\ \mathcal{L}_0\} \xrightarrow{e_1^{\mathrm{LAW}} = e_2^{\mathrm{LAW}}} \{\mathit{true}\}\ G_s; \theta_{sr}(\theta'(G')); G_r\ \{\theta_{sr}(l') = \mathbf{in}\ \theta_{sr}(e') \wedge \mathit{modifies}\ \mathcal{L}_s\}}$$

Figure 2. Generating a candidate

the control-flow graph G that implements the RTL, plus a postcondition that gives the source and destination of the assignment.

A graph and a postcondition suffice to express the solutions to our problem—the implementations of the tiles we are trying to discover. But this form is not general enough to enable us to reason about composition of machine instructions, which requires us to reason about intermediate states. We therefore use Hoare triples:

$$\{P\}\ G\ \{Q\}$$

The Hoare triple says that executing graph G in an input state satisfying precondition P results in an output state satisfying postcondition Q. The postcondition may refer to both states: to refer to the contents of a location l in the output state, we write just l; to refer to the contents of that same location in the input state, we write $\mathbf{in}\ l$. We lift $\mathbf{in}$ to expressions using the obvious homomorphism.

Because we are interested in implementing tiles, and because a tile must establish its postcondition regardless of machine state, the precondition of a Hoare triple in the pool is always true. Because each tile implements a single assignment, we also restrict postconditions to the following canonical form:

$$l = \mathbf{in}\ e \wedge \mathit{modifies}\ \mathcal{L}$$

The postcondition describes the contents of an *output-state* location l in terms of the value of an *input-state* expression. The postcondition also says what "extra" locations $\mathcal{L}$ are modified by the graph; this set, which does not contain l, accounts for unwanted assignments. If a location is not in the set $\mathcal{L} \cup \{l\}$, its value remains unchanged. As an example, the PowerPC implementation of the bitwise-complement tile in Section 6.1 is described by this triple:

$$\{\mathit{true}\}\ G_{com}\ \{r_1 = \mathbf{in}\ \neg r_2 \wedge \mathit{modifies}\ \{t\}\}$$

7.1 How new candidate Hoare triples are created

Our search algorithm looks for Hoare triples that implement new assignments. Its fundamental operation is to use an algebraic law $e_1^{\mathrm{LAW}} = e_2^{\mathrm{LAW}}$ to rewrite the postcondition of a Hoare triple that is already in the pool, producing a new, *candidate* Hoare triple with a new postcondition. We write the operation using this judgment:

$$\{\mathit{true}\}\ G\ \{Q\} \xrightarrow{e_1^{\mathrm{LAW}} = e_2^{\mathrm{LAW}}} \{\mathit{true}\}\ G'\ \{Q'\}.$$

A derivation of this judgment, which always ends in rule CREATE CANDIDATE from Figure 2, is constructed as follows:

1. Choose a Hoare triple $\{\mathit{true}\}\ G_0\ \{l_0 = \mathbf{in}\ e_0 \wedge \mathit{modifies}\ \mathcal{L}_0\}$.
2. Choose a subexpression e of e_0 such that $e_0 = E[e]$.
3. Choose an algebraic law $e_1^{\mathrm{LAW}} = e_2^{\mathrm{LAW}}$.
4. Use the algebraic law to rewrite e. Specifically, in the precondition, we assume that $e = e_1^{\mathrm{LAW}}$ and use that assumption to rewrite the postcondition:

   $$\{e = e_1^{\mathrm{LAW}}\}\ G_0\ \{l_0 = \mathbf{in}\ E[e_2^{\mathrm{LAW}}] \wedge \mathit{modifies}\ \mathcal{L}_0\}.$$

   Because this new Hoare triple has a nontrivial precondition, we can't use it right away.
5. To use the new triple, we try to discharge the precondition $e = e_1^{\mathrm{LAW}}$ by running an algorithm we call *establishment*. Establishment tries to match e and e_1^{LAW} in a manner akin to unification, so one of the results is a substitution θ'. A substitution maps operand metavariables $x_{\in ls}$ to locations and $x_{\#w}$ to compile-time constant expressions:

   $$\theta ::= \cdot \mid \theta[x_{\in ls} \mapsto l] \mid \theta[x_{\#w} \mapsto e]$$

   We write $\cdot$ for the identity substitution; square brackets stand for function composition. For example, $\theta[x_{\in ls} \mapsto l]$ first applies θ, then substitutes l for $x_{\in ls}$ in the result.

   Unification alone cannot establish the truth of equations that refer to the contents of machine locations. Given the triple $\{e = e_1^{\mathrm{LAW}}\}\ G_0\ \{Q\}$, our algorithm therefore produces not only a substitution θ' but also a new graph $G' = G_{est}; G_0$, where G_{est} alters machine state in order to make $e = e_1^{\mathrm{LAW}}$. We write the result

   $$\theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\},$$

   and we call this form an *extended Hoare triple*. An extended Hoare triple is valid whenever applying θ' to precondition, graph, and postcondition produces a valid Hoare triple. We keep the substitution separate, to the left of a turnstile, in order to detect aliasing when a machine instruction is used. (See the final premise of rule USE RESULT in Figure 4.)

   Establishment takes an extended Hoare triple as a goal, and when successful, it produces a new extended Hoare triple whose precondition is trivially satisfied. We write the relation between goal and result using the arrow $\overset{est}{\rightsquigarrow}$:

   $$\theta \vdash \{P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}.$$

   In any machine state, executing graph $\theta'(G')$ puts the machine into a state satisfying the postcondition $\theta'(Q')$. The inference rules for $\overset{est}{\rightsquigarrow}$ are given in Figure 4 and explained in Section 7.3.
6. In a candidate triple, it is not enough that substitution θ' and graph G' establish postcondition $l' = \mathbf{in}\ e' \wedge \mathit{modifies}\ \mathcal{L}'$, because $\mathcal{L}'$ may include observable locations that should not be modified, such as machine registers. If this is so, we examine $\mathcal{L}'$ and find a substitution θ_{sr} and graphs G_s and G_r such that only locations in set $\mathcal{L}_s$, which are unobservable, are modified. We write $\langle G_s, G_r, \mathcal{L}_s, \theta_{sr}\rangle \Updownarrow \mathrm{observable}(\mathcal{L}')$. Substitution θ_{sr} renames observable operand metavariables in $\mathcal{L}'$ to unobservable fresh temporaries. If observable locations remain, which rarely happens, graph G_s saves their contents to temporaries or scratch locations in $\mathcal{L}_s$, and G_r restores the original values from $\mathcal{L}_s$. The graph $G_s; \theta_{sr}(\theta'(G')); G_r$ therefore makes modifications to locations in $\mathcal{L}'$ unobservable.
7. The candidate Hoare triple is created by using the graph $G_s; \theta_{sr}(\theta'(G')); G_r$ computed in Step 6 and the postcondition derived by establishment in Step 5. Only θ_{sr} is applied to the postcondition; by construction, no metavariables from θ' appear in the postcondition derived by establishment.

INSTANTIATE MACHINE CODE

$$\frac{\begin{array}{c} l \notin \mathit{locs}(e) \qquad \{\mathit{true}\}\ G\ \{l = \mathbf{in}\ e \wedge \mathit{modifies}\ \mathcal{L}\} \in \mathit{Pool} \\ \theta = \theta_i \circ \theta_{\mathit{fresh}} \qquad \theta_{\mathit{fresh}} \text{ freshens metavariables} \end{array}}{\vdash_{\mathit{inst}}\ \{\mathit{true}\}\ \theta(G)\ \{\theta(l) = \mathbf{in}\ \theta(e) \wedge \mathit{modifies}\ \theta(\mathcal{L})\}}$$

INSTANTIATE MACHINE CODE WITH MOVE

$$\frac{\begin{array}{c} l \in \mathit{locs}(e) \qquad \{\mathit{true}\}\ G\ \{l = \mathbf{in}\ e \wedge \mathit{modifies}\ \mathcal{L}\} \in \mathit{Pool} \\ \vdash_{\mathit{inst}}\ \{\mathit{true}\}\ G'\ \{l = \mathbf{in}\ l' \wedge \mathit{modifies}\ \mathcal{L}'\} \\ l \notin \mathit{locs}(l') \qquad \theta = \theta_i \circ \theta_{\mathit{fresh}} \qquad \theta_{\mathit{fresh}} \text{ freshens metavariables} \end{array}}{\vdash_{\mathit{inst}}\ \{\mathit{true}\}\ \theta(G'; G)\ \{\theta(l) = \mathbf{in}\ \theta(e[l \mapsto l']) \wedge \mathit{modifies}\ \theta(\mathcal{L} \cup \mathcal{L}')\}}$$

Figure 3. Instantiation of code from pool

ALL SATISFIED

$$\frac{}{\theta \vdash \{\mathit{true}\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta \vdash \{\mathit{true}\}\ G\ \{Q\}}$$

EQUAL EXPRESSIONS

$$\frac{\theta \vdash \{P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}}{\theta \vdash \{e = e \wedge P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}}$$

SYMMETRY

$$\frac{\theta \vdash \{e' = e \wedge P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}}{\theta \vdash \{e = e' \wedge P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}}$$

SPECIALIZE LAW

$$\frac{\begin{array}{c} e \text{ has no law metavariables} \qquad x_L \notin \mathit{fv}(G) \\ \theta \vdash \{P[x_L \mapsto e]\}\ G\ \{Q[x_L \mapsto e]\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\} \end{array}}{\theta \vdash \{x_L = e \wedge P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}}$$

MATCH LOCATION

$$\frac{\begin{array}{c} l \in ls \qquad x_{\in ls} \notin \mathrm{dom}(\theta) \qquad \theta_{\mathit{ext}} = \theta[x_{\in ls} \mapsto l] \\ \theta_{\mathit{ext}} \vdash \{P[x_{\in ls} \mapsto l]\}\ G\ \{Q[x_{\in ls} \mapsto l]\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\} \end{array}}{\theta \vdash \{x_{\in ls} = l \wedge P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}}$$

MATCH COMPILE-TIME CONSTANT EXPRESSION

$$\frac{\begin{array}{c} e \text{ is computable at compile time and fits in } w \text{ bits} \qquad x_{\#w} \notin \mathit{fv}(e) \\ \mathit{shrinks}(x_{\#w} = e \wedge P) \qquad x_{\#w} \notin \mathrm{dom}(\theta) \qquad \theta_{\mathit{ext}} = \theta[x_{\#w} \mapsto e] \\ \theta_{\mathit{ext}} \vdash \{P[x_{\#w} \mapsto e]\}\ G\ \{Q[x_{\#w} \mapsto e]\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\} \end{array}}{\theta \vdash \{x_{\#w} = e \wedge P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}}$$

REDUCE APPLICATIONS

$$\frac{\theta \vdash \{e_1 = e'_1 \wedge \cdots \wedge e_n = e'_n \wedge P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}}{\theta \vdash \{\oplus(e_1, \ldots, e_n) = \oplus(e'_1, \ldots, e'_n) \wedge P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}}$$

PREPARE OPERATOR

$$\frac{\begin{array}{c} \vdash_{\mathit{inst}}\ \{\mathit{true}\}\ G_p\ \{l' = \mathbf{in}\ \oplus(e'_1, \ldots, e'_n) \wedge \mathit{modifies}\ \mathcal{L}\} \\ \theta \vdash \{l = l' \wedge l' = \oplus(e'_1, \ldots, e'_n) \wedge \bigwedge_{1 \le i \le n} e'_i = e_i \wedge P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\} \end{array}}{\theta \vdash \{l = \oplus(e_1, \ldots, e_n) \wedge P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}}$$

PREPARE CONSTANT

$$\frac{\begin{array}{c} e \text{ and } e' \text{ are computable at compile time} \\ \vdash_{\mathit{inst}}\ \{\mathit{true}\}\ G_p\ \{l' = \mathbf{in}\ e' \wedge \mathit{modifies}\ \mathcal{L}\} \qquad \theta \vdash \{l = l' \wedge l' = e' \wedge e' = e \wedge P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\} \end{array}}{\theta \vdash \{l = e \wedge P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}}$$

USE RESULT

$$\frac{\begin{array}{c} \vdash_{\mathit{inst}}\ \{\mathit{true}\}\ G_p\ \{l = \mathbf{in}\ e \wedge \mathit{modifies}\ \mathcal{L}\} \qquad l \notin \mathit{locs}(e) \qquad \mathit{shrinks}(l = e \wedge P) \qquad \mathcal{L}_1 = \mathcal{L}_0 \cup \mathcal{L} \cup \{l\} \setminus \{l_0\} \\ \theta \vdash \{P[l \mapsto e]\}\ G_p; G\ \{l_0 = \mathbf{in}\ e_0[l \mapsto e] \wedge \mathit{modifies}\ \mathcal{L}_1\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\} \qquad \theta'(\mathcal{L} \cup \{l\}) \cap \theta'(\mathit{locs}(e_0) \setminus \{l\}) = \emptyset \end{array}}{\theta \vdash \{l = e \wedge P\}\ G\ \{l_0 = \mathbf{in}\ e_0 \wedge \mathit{modifies}\ \mathcal{L}_0\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}}$$

Figure 4. Rules for establishment: $\theta \vdash \{P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}$

7.2 Example of finding a new Hoare triple

We now give a full formal treatment of the bitwise-complement example from Section 6.1. In the process we explain many of the rules for deriving Hoare triples, as well as our algorithmic search for derivations. We begin with this pool:[2]

$$\{\mathit{true}\}\ G_{\mathit{ldzero}}\ \{r'_1 = \mathbf{in}\ 0 \wedge \mathit{modifies}\ \emptyset\}$$
$$\{\mathit{true}\}\ G_{\mathit{bitnor}}\ \{r_1 = \mathbf{in}\ \neg(r_2 \vee r_3) \wedge \mathit{modifies}\ \emptyset\}$$

We choose the triple with graph G_{bitnor} and the algebraic law $x \vee 0 = x$. From rule CREATE CANDIDATE in Figure 2, we choose subexpression $r_2 \vee r_3$ as e, so $E[\,]$ is $\neg[\,]$. We can apply the algebraic law if we can discharge the precondition $r_2 \vee r_3 = x \vee 0$, by using the rules in Figure 4 to derive

$$\cdot \vdash \{r_2 \vee r_3 = x \vee 0\}\ G_{\mathit{bitnor}}\ \{r_1 = \mathbf{in}\ \neg x \wedge \mathit{modifies}\ \emptyset\}$$
$$\overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}$$

Logically, $\overset{est}{\rightsquigarrow}$ relates a valid extended Hoare triple $\theta \vdash \{P\}\ G\ \{Q\}$ to a new extended Hoare triple $\theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}$. Algorithmically, we start with the identity substitution and a precondition P, and we search for a substitution θ' and graph $G' = G_{est}; G$ such that when we substitute for metavariables as specified by θ', executing $\theta'(G')$ establishes postcondition Q', where Q' is derived from Q by substitution for both metavariables and machine loca-

[2] We write the load-zero triple using metavariable r'_1 in order to emphasize that when we use a triple in the pool, we freshen all its metavariables.

$$\text{Reduce apps}\ \dfrac{\text{Symmetry}\ \dfrac{\text{Specialize law}\ \dfrac{\text{Use result}\ \dfrac{\text{Inst. mach. code}\ \dfrac{\begin{array}{c}\{\mathit{true}\}\ r_1' := 0\ \{r_1' = \mathbf{in}\,0 \wedge \mathit{modifies}\ \emptyset\} \in \mathit{Pool} \\ r_3 \notin \emptyset \qquad \theta = [r_1' \mapsto r_3]\end{array}}{\vdash_{inst}\ \{\mathit{true}\}\ r_3 := 0\ \{r_3 = \mathbf{in}\,0 \wedge \mathit{modifies}\ \emptyset\}} \qquad r_3 \notin \emptyset \qquad \mathit{shrinks}(r_3 = 0 \wedge \mathit{true}) \qquad \{r_3\} \cap \{r_2\} = \emptyset}{\cdot \vdash \{r_3 = 0\}\ G_{bitnor}\ \{r_1 = \mathbf{in}\,\neg r_2 \wedge \mathit{modifies}\ \emptyset\} \overset{est}{\rightsquigarrow} \cdot \vdash \{\mathit{true}\}\ r_3 := 0; G_{bitnor}\ \{r_1 = \mathbf{in}\,\neg r_2 \wedge \mathit{modifies}\ \{r_3\}\}} \qquad r_2 \text{ has no law metavariables} \qquad x \notin \{r_1, r_2, r_3\}}{\cdot \vdash \{x = r_2 \ \ \wedge \ \ r_3 = 0\}\ G_{bitnor}\ \{r_1 = \mathbf{in}\,\neg x \wedge \mathit{modifies}\ \emptyset\} \overset{est}{\rightsquigarrow} \cdot \vdash \{\mathit{true}\}\ r_3 := 0; G_{bitnor}\ \{r_1 = \mathbf{in}\,\neg r_2 \wedge \mathit{modifies}\ \{r_3\}\}}}{\cdot \vdash \{r_2 = x \ \ \wedge \ \ r_3 = 0\}\ G_{bitnor}\ \{r_1 = \mathbf{in}\,\neg x \wedge \mathit{modifies}\ \emptyset\} \overset{est}{\rightsquigarrow} \cdot \vdash \{\mathit{true}\}\ r_3 := 0; G_{bitnor}\ \{r_1 = \mathbf{in}\,\neg r_2 \wedge \mathit{modifies}\ \{r_3\}\}}}{\cdot \vdash \{r_2 \vee r_3 = x \vee 0\}\ G_{bitnor}\ \{r_1 = \mathbf{in}\,\neg x \wedge \mathit{modifies}\ \emptyset\} \overset{est}{\rightsquigarrow} \cdot \vdash \{\mathit{true}\}\ r_3 := 0; G_{bitnor}\ \{r_1 = \mathbf{in}\,\neg r_2 \wedge \mathit{modifies}\ \{r_3\}\}}$$

Figure 5. Derivation of a graph that implements $r_1 := \neg r_2$

tions. An invariant of the algorithm is that precondition P is a conjunction of equalities. The algorithm finds a sequence of graphs that establishes the equalities; the last graph in the sequence establishes the first equality in the conjunction. From the logical perspective, equalities $e = e'$ and $e' = e$ are equivalent, but in order to guarantee termination, our search for a derivation distinguishes them (Section 7.3.3).

Our algorithm proceeds by structural induction on the precondition $r_2 \vee r_3 = x \vee 0$, eventually producing the derivation shown in Figure 5. The first (and only) equality in the precondition applies operator $\vee$ on both sides, and the rule with this syntactic form in its conclusion is REDUCE APPLICATIONS.[3] The rule converts equality of application to conjunction of equalities, so our new precondition is $r_2 = x \wedge r_3 = 0$. At this point, we must apply SYMMETRY to make progress, and we get precondition $x = r_2 \wedge r_3 = 0$. Now only one rule applies: SPECIALIZE LAW. We substitute r_2 for x in the rest of the precondition and in the postcondition, which gives us the new Hoare triple $\{r_3 = 0\}\ G_{bitnor}\ \{r_1 = \mathbf{in}\,\neg r_2 \wedge \mathit{modifies}\ \emptyset\}$ for the inductive case. SPECIALIZE LAW does not extend θ, because θ tracks substitutions for operand metavariables only.

To establish precondition $r_3 = 0$, we use graph $r_3 := 0$ from the pool, applying rule USE RESULT. This rule is the only rule that adds code from the pool to the graph in progress, and it requires a new judgment for instantiating a graph from the pool:

$$\vdash_{inst}\ \{\mathit{true}\}\ G\ \{Q\}$$

Our example uses rule INSTANTIATE MACHINE CODE (Figure 3): we make metavariables fresh using a substitution θ_{fresh}, and then to match the postcondition Q, we may rebind the fresh metavariables using an additional substitution θ_i. In our example, the precondition is $r_3 = 0$ so (ignoring $\mathit{modifies}$ clauses) we want a $\vdash_{inst}$ judgment with postcondition $r_3 = \mathbf{in}\,0$. The code in the pool has postcondition $r_1' = \mathbf{in}\,0$. Metavariable r_1' must be replaced with a fresh metavariable, but we can then choose θ_i to replace that metavariable with r_3, enabling ourselves to conclude

$$\vdash_{inst}\ \{\mathit{true}\}\ r_3 := 0\ \{r_3 = \mathbf{in}\,0 \wedge \mathit{modifies}\ \emptyset\}$$

The remaining precondition is the empty conjunction true, so we apply ALL SATISFIED. (To avoid clutter, we omit the application from Figure 5.)

We now complete the derivation, filling in the right-hand side of each judgment; each rule derives the same substitution and Hoare triple as its inductive case. We also check the final premise of USE RESULT, here verifying (by $\{r_3\} \cap \{r_2\} = \emptyset$) that the assignment to r_3 does not affect the value of $\neg r_2$. The result of the derivation is the conclusion

$$\begin{array}{l}\cdot \vdash \{r_2 \vee r_3 = x \vee 0\}\ G_{bitnor}\ \{r_1 = \mathbf{in}\,\neg x \wedge \mathit{modifies}\ \emptyset\} \\ \quad \overset{est}{\rightsquigarrow} \cdot \vdash \{\mathit{true}\}\ r_3 := 0; G_{bitnor}\ \{r_1 = \mathbf{in}\,\neg r_2 \wedge \mathit{modifies}\ \{r_3\}\}\end{array}$$

When r_3 is replaced by a temporary t, graph $t := 0; G_{bitnor}[r_3 \mapsto t]$ will be a candidate for addition to the pool.

7.3 Our rules for deriving Hoare triples

7.3.1 Instantiation

Our example applies rule INSTANTIATE MACHINE CODE to use an instruction from the pool. But an instruction with postcondition $l' = \mathbf{in}\,e'$ can be used to establish $l' = e'$ only if $l' \notin \mathit{locs}(e')$, where $\mathit{locs}(e')$ are the locations mentioned in e'. But on a real machine, l' may appear in $\mathit{locs}(e')$; that is, an instruction may insist on writing one of the locations it reads. (See the *x*86 or any other two-address instruction set.) To exploit such instructions, we provide a second way of deriving the $\vdash_{inst}$ judgment, INSTANTIATE MACHINE CODE WITH MOVE. Given a triple from the pool that computes $l := e$, where $l \in \mathit{locs}(e)$, this rule constructs a sequence of instructions to replace every use of l in e with a different location l'.

7.3.2 Soundness

Given a set of algebraic laws and a pool of Hoare triples, our algorithm finds new Hoare triples which are candidates for addition to the pool. Provided the algebraic laws are sound and the pool contains only valid Hoare triples, the rules for establishment are sound. That is, if $\theta \vdash \{P\}\ G\ \{Q\} \overset{est}{\rightsquigarrow} \theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}$, and if $\theta \vdash \{P\}\ G\ \{Q\}$ is valid, then $\theta' \vdash \{\mathit{true}\}\ G'\ \{Q'\}$ is also valid. To simplify the proof of soundness, we have carefully engineered the rules so that only USE RESULT actually *uses* code from the pool. We prove soundness by induction on the height of derivations.

- The base case is ALL SATISFIED; the rule is sound because the right-hand side of the conclusion is valid by hypothesis.
- The most interesting rule is USE RESULT. By design, it is the only rule that uses machine code (G_p) to satisfy a precondition. Its soundness is based on Hoare's axiom for assignment, but it may be easier to see that the new precondition $P[l \mapsto e]$ is the weakest precondition of $l := e$ with respect to P. In the inductive case, the substitution $e_0[l \mapsto e]$ is correct provided that the only location in e_0 which G_p mutates is l. This provision is enforced by the side condition $\theta'(\mathcal{L} \cup \{l\}) \cap \theta'(\mathit{locs}(e_0) \setminus \{l\}) = \emptyset$, which in our example was $\{r_3\} \cap \{r_2\} = \emptyset$. We have to apply the substitution θ' because establishment might cause two otherwise distinct locations to alias.
- Rules EQUAL EXPRESSIONS, SYMMETRY, REDUCE APPLICATIONS, PREPARE OPERATOR, and PREPARE CONSTANT are variations on a theme: replace a goal P_1 with a smaller or better goal P_2.

[3] SYMMETRY matches any syntactic form, but it is used only to enable progress when progress is not otherwise possible.

Initial pool:		
A:	$r_1 := \mathtt{sx}(\mathtt{lobits}_{16}(k_{32}))$	load signed k_{16}
B:	$r_1 := r_2 \vee \mathtt{zx}(\mathtt{lobits}_{16}(k_{32}))$	bitwise-or immediate
C:	$r_1 := r_2 \vee (k_{32} \gg_l 16) \ll 16$	load upper immediate
Round 1 finds load-immediate 0 and or-with-0:		
D:	$r_1 := \mathtt{sx}(\mathtt{lobits}_{16}(0))$	$\{r_1 := 0\}$
E:	$r_1 := r_2 \vee \mathtt{zx}(\mathtt{lobits}_{16}(0))$	$\{r_1 := r_2 \vee 0\}$
Round 2 finds register-register move and zero-extend-low-bits:		
F:	$r_1 := r_2 \vee \mathtt{zx}(\mathtt{lobits}_{16}(0))$	$\{r_1 := r_2\}$
G:	$t_1 := \mathtt{sx}(\mathtt{lobits}_{16}(0));$ $r_1 := t_1 \vee \mathtt{zx}(\mathtt{lobits}_{16}(k_{32}))$	$\{r_1 := \mathtt{zx}(\mathtt{lobits}_{16}(k_{32}))\}$
Round 3 finds load-immediate of a 32-bit k:		
H:	$t_1 := \mathtt{sx}(\mathtt{lobits}_{16}(0));$ $t_2 := t_1 \vee \mathtt{zx}(\mathtt{lobits}_{16}(k_{32}))$ $r_1 := t_2 \vee (k_{32} \gg_l 16) \ll 16;$	$\{r_1 := k_{32}\}$

Figure 6. Finding instructions by rounds on the PowerPC. Modifies clauses are omitted to save space.

These rules are sound because if $\theta \vdash \{P_1\}\ G\ \{Q\}$ is valid and if P_2 implies P_1, then $\theta \vdash \{P_2\}\ G\ \{Q\}$ is valid. By induction, the extended triple $\theta' \vdash \{true\}\ G'\ \{Q'\}$ derived in the inductive case must also be valid.

The PREPARE rules avoid using code from the pool; instead, they set up a precondition that is designed to be discharged by USE RESULT. This trick simplifies the proof of soundness, at the cost of making the proof of termination slightly more complicated.

- The metavariable rules SPECIALIZE LAW, MATCH LOCATION, and MATCH COMPILE-TIME CONSTANT EXPRESSION all introduce substitutions. They are sound because if we apply a substitution to a valid triple, the result is also a valid triple.

7.3.3 Termination of establishment

It is easy to write a terminating algorithm that searches for derivations. During the search, we say that a precondition *gets smaller* if the number of law metavariables x_L decreases, or if the expressions on the right-hand sides get smaller, or if the number of conjuncts decreases. (The precise condition involves counting the number of conjuncts with right-hand sides of height n, for each n.) In most inference rules, the inductive case obviously uses a smaller precondition. But USE RESULT and MATCH COMPILE-TIME CONSTANT require side conditions to ensure the new precondition is smaller: in USE RESULT, condition $shrinks(l = e \wedge P)$ ensures that the new goal $P[l \mapsto e]$ is no larger than P. A similar condition applies to MATCH COMPILE-TIME CONSTANT. We define $shrinks(l = e \wedge P)$ to hold unless e is an application and l appears free in the right-hand side of an equality in P.

SYMMETRY does not reduce the goal, but we use it only to move x_L or $x_{\#w}$ from the right-hand side of a subgoal to the left, after which search makes progress or fails.

PREPARE OPERATOR and PREPARE CONSTANT *temporarily* increase the goal, but in both rules, the second subgoal of the new goal will be discharged by USE RESULT. The idea is to see (via the instantiation judgment $\vdash_{inst}$) if code in the pool applies the operator or computes the compile-time constant expression in the precondition. If so, the PREPARE rules take the postcondition $l' = \textbf{in}\ e'$ established by the code in the pool, and they try a new goal whose precondition includes the equality $l' = e'$, knowing that this equality will be discharged by USE RESULT.

```
1  Initialize $Pool$ with Hoare triples from machine description
2  repeat until $Pool$ stops growing
3    Build data-movement graph from current contents of $Pool$
4    Add new data-movement edges to $Pool$ as Hoare triples
5    foreach $\{true\}\ G_0\ \{l_0 = \textbf{in}\ e_0 \wedge modifies\ \mathcal{L}\} \in Pool$
6      foreach subexpression and context $E[e] = e_0$
7        foreach law $e_1^{\mathrm{LAW}} = e_2^{\mathrm{LAW}}$
8          if $\{true\}\ G_0\ \{l_0 = \textbf{in}\ E[e] \wedge modifies\ \mathcal{L}_0\} \xrightarrow{e_1^{\mathrm{LAW}} = e_2^{\mathrm{LAW}}}$
9              $\{true\}\ G'\ \{l' = \textbf{in}\ e' \wedge modifies\ \mathcal{L}'\}$
10         and $l' = \textbf{in}\ e'$ passes our utility test (Section 9)
11         then $Pool$ += $\{true\}\ G'\ \{l' = \textbf{in}\ e' \wedge modifies\ \mathcal{L}'\}$
```

Figure 7. Pseudocode summarizing our algorithm.

7.4 Adding new Hoare triples in rounds

Above, we show how by choosing an existing Hoare triple, an algebraic law, and a subexpression, we can search for a new Hoare triple to add to the pool. Here, we show what we do with the triples we find. After initializing the pool with triples derived from the machine description, we add new triples in *rounds*. Hoare triples found in one round can be used to find new Hoare triples in later rounds, as shown in Figure 6.

Figure 6 shows the control-flow graphs of eight example triples from the PowerPC. The three triples in the initial pool come directly from machine instructions: load 16-bit immediate, bitwise-or immediate, and load upper immediate, labelled A, B, and C respectively. In the first round, applying the law $\mathtt{sx}(\mathtt{lobits_n}(0)) = 0$ to triple A produces an implementation of load zero (D); and applying the law $\mathtt{zx}(\mathtt{lobits_n}(0)) = 0$ to triple B produces an implementation computing the bitwise-or of a register and zero (E). In the second round, applying the law $x_L \vee 0 = x_L$ to triple E produces an implementation of register-register move (F); and applying the law $0 \vee x_L = x_L$ to triple B results in a sequence that combines triples D and B to compute the zero-extension of the least significant 16 bits of an immediate value (G). Neither of these laws can be applied in the first round because we need to use triples that are produced by the first round. The interesting round is the third round, where we apply the law $\mathtt{zx}(\mathtt{lobits}_n(x_L)) \vee ((x_L \gg_l n) \ll n) = x_L$ to triple C, which results in a sequence (H) that combines triples G and C to load a full 32-bit literal constant: it implements the load-immediate tile.

Our search for triples in rounds is implemented by the algorithm in Figure 7. The first step (line 1) is to initialize the pool using RTLs from the machine description. Because not all RTLs are useful for implementing tiles, we add only those that pass a utility test, which we define in Section 9. For each *useful* assignment $l_i := e_i$ of an RTL $l_1 := e_1 \mid \cdots \mid l_n := e_n$, we add the following Hoare triple to the pool:

$$\{true\}\ l_1 := e_1 \mid \cdots \mid l_n := e_n\ \{l_i = \textbf{in}\ e_i \wedge modifies \bigcup_{j \neq i} \{l_j\}\}$$

We then begin the first round of search.

A round is implemented by the **repeat** loop on line 2 of Figure 7, which keeps searching for new candidates until no more Hoare triples are added to the pool. Before the search begins, we use the data-movement triples in the pool to build a new data-movement graph, on line 3. (Because each round may discover new data-movement triples, the graph from the previous round may be obsolete.) Any new edges are added to the pool (line 4).

The search itself tries all possible combinations of triples in the pool, algebraic laws, and subexpressions (lines 5–7). For efficiency, candidate laws are chosen *after* subexpressions (line 7): when we know that the goal is to establish $e = e_1^{\mathrm{LAW}}$, then if e is an application of an operator, we need consider only laws in which e_1^{LAW} is an application of the same operator.

Having made our choices, we attempt to construct a derivation (lines 8–9) that ends in an application of the CREATE CANDIDATE rule from Figure 2 in Section 7.1. If we find a new triple, we can add it to the pool, *provided* it passes our utility test (line 10). The new triple is equivalent to the assignment $l' := e'$, with additional modifications to unobservable locations. If this assignment appears useful, we add the new Hoare triple to the pool for use in the next round (line 11); otherwise, we discard it.

By controlling which Hoare triples are added to the pool, our utility test guarantees that the number of rounds is bounded. Because the argument about termination is not trivial, we discuss it in its own section (Section 9).

8. Finding implementations of tiles is undecidable

Ideally there would be a decision procedure that would enable us both to limit the number of Hoare triples added to the pool and to guarantee that every potentially useful Hoare triple is added. Unfortunately, the underlying problem is undecidable: no terminating algorithm can guarantee to find implementations of all implementable tiles. The problem of finding an implementation of a tile is closely related to the more general, undecidable problem of determining if two programs are equivalent.

A sequence of instructions implements a tile if, using some set of algebraic laws, we can show that the RTL computed by the sequence of instructions is equivalent to the RTL that represents the tile. For convenience, we work not with algebraic laws, but with rewrite rules; an algebraic law $e = e'$ can be expressed as a pair of rewrite rules $e \to e'$ and $e' \to e$.

THEOREM 1. *There is no algorithm that, given an arbitrary set of rewrite rules and RTLs R_1 and R_2, can decide if $R_1 \to^* R_2$.*

Due to space restrictions, we only sketch the proof; a full proof is given by Dias (2008, Appendix A). The proof proceeds by reduction from the halting problem; we adapted well-known proofs of undecidability for strong normalization in term-rewriting systems (Bezem, Klop, and de Vrijer 2003).

The reduction defines an embedding from a Turing-machine configuration to an RTL, and an embedding from the Turing-machine's transition function to rewrite rules. We define a deterministic Turing machine as a triple $\langle Q, S, \delta \rangle$, where Q is a finite set of states, S is the finite set of symbols that may be written on the Turing machine's tape (including the blank symbol $\Box$), and δ is the transition function $Q \times S \to Q \times S \times \{L, R\}$, where L and R stand for "left" and "right."

The model of the Turing machine is that there is a tape of infinite length in both directions, and at each position on the tape is a symbol. At any given time, the Turing machine is in some state q at a position p on the tape. The machine takes a step by reading the symbol s at position p and calling the transition function $\delta(q, s)$, which returns the new state of the machine, the new symbol to write at position p, and the direction to move along the tape. The configuration of the Turing machine is represented by the tuple $\langle q, T_l, s, T_r \rangle$, where q is the current state, T_l is the stack of symbols to the left, s is the current symbol, and T_r is the stack of symbols to the right.

To embed a Turing-machine configuration as an RTL, we encode each state q as a ternary operator and each symbol s as a nullary operator. We use the literal 0 to represent an empty stack of blank symbols, and we use the infix binary operator : as a *cons* operator to construct stacks. For example, a Turing-machine configuration in state q_1 with symbols s_1 and s_2 to the left, symbol s_3 at the current position, and symbol s_4 to the right, can be encoded using the RTL $l_0 := q_1(s_1 : s_2 : 0, s_3, s_4 : 0)$ where l_0 is defined as a location on the machine.

The embedding of the Turing-machine transition function to rewrite rules is complicated by the fact that the embedding of Turing-machine configuration may produce either an empty or a nonempty stack. Consequently, each transition is embedded as a pair of rewrite rules. For example, the embedding of a transition to the left $\delta(q, s) = (q', s', L)$ produces two rewrite rules:

$$q(x_1 : x_2, s, x_3) \to q'(x_2, x_1, s' : x_3)$$

$$q(0, s, x) \to q'(0, \Box, s' : x)$$

We also define a distinguished halting RTL: $l_0 := \texttt{halted}()$, where `halted` is a unique, nullary RTL operator. The reduction is completed by adding a rewrite rule to ensure that the embedding of any halting state in the Turing machine can be rewritten to the distinguished halting RTL. The Turing machine halts if the transition function is not defined on a pair (q, s). For each such pair $(q, s) \notin dom(\delta)$, we add the rewrite rule

$$q(x, s, x') \to \texttt{halted}()$$

Given this embedding, our proof shows that there is a bisimulation between the transitions of the Turing machine and the rewrite rules produced by the embedding of the transition function. Using the bisimulation result, we show that an algorithm that decides RTL equivalence could be used to solve the halting problem by embedding the starting configuration of the Turing machine in an RTL R and using the algorithm to check if R is equivalent to the halting RTL. Because the halting problem is undecidable, we conclude that there can be no algorithm that decides RTL equivalence.

This undecidability result holds for the general problem, where we can choose an arbitrary input instruction set, an arbitrary set of tiles, and an arbitrary set of algebraic laws. An important question remains open: is there a *limited* set of algebraic laws, expressive enough to find implementations of all the tiles for a well-defined set of machines, but for which the problem of finding tiles is decidable?

9. Termination and our utility test

Because the general problem is undecidable, we use a heuristic to restrict what Hoare triples can be added to the pool. We have found a heuristic with excellent properties:

- On real machines, we are likely to find implementations of all the tiles.
- The pool stays small enough that the tileset generator runs in minutes, not hours.

The discovery of this heuristic is one of the significant contributions of this paper. Before describing it, we explain why more typical heuristics are not suitable.

One obvious heuristic would be to limit the number of machine instructions used in a graph G. Another would be to limit the maximum height of any expression appearing in a postcondition. Because these heuristics make decisions based on size alone, not using any information about whether the RTL might help implement a tile, they are unsuitable in two ways. First, they are likely to keep useless implementations that are the right size but will never help implement a tile. Second, they are likely to reject useful implementations: to implement some tiles, long sequences of

```
1 $estimateLaws(\{true\}\ G\ \{l = \mathbf{in}\ e \wedge modifies\ \mathcal{L}\})$
2   $\mathbf{return}\ expEstimate(e)$
3
4 $expEstimate(e) : e \rightarrow int$
5   $\mathbf{if}$ $e$ is computed by an expansion tile $\mathbf{then\ return}\ 0$
6   $\mathbf{else\ return}\ \min\{1 + coverWithLawFragment(e, e')$
7                                  $\mid E[e'] = e_2 \in$ Laws$\}$
8
9 $coverWithLawFragment(e, e') : e \times e \rightarrow int$
10  $\mathbf{case}\ (e,\ e')\ \mathbf{of}$:
11    $(\oplus(e_1, \ldots, e_n),\ \oplus(e'_1, \ldots, e'_n))$:
12       $\mathbf{return}\ \sum_{i=1}^{n} \min(coverWithLawFragment(e_i, e'_i),$
13                          $expEstimate(e_i))$
14    $\mathbf{default}$: $\mathbf{return}\ \infty$
```

Figure 8. Our heuristic estimates the minimum number of algebraic laws that might be applied before our algorithm can conclude that the input Hoare triple is used in the implementation of a tile.

instructions and tall expressions may be necessary. For example, as Figure 6 shows, a load-immediate tile for a RISC machine typically requires multiple instructions and an expression $((k \gg_l n) \ll n) \vee \mathtt{zx}(\mathtt{lobits}_n(k))$ of height 4. For these reasons, we abandoned traditional heuristics.

9.1 Our new heuristic: pruning according to predicted utility

We have developed a new heuristic which limits the pool to hold only those Hoare triples deemed *likely to be useful in implementing some expansion tile*. Our utility test is inspired by the technique we use to find new implementations: the application of algebraic laws. We hypothesize that when more algebraic laws have to be applied to a Hoare triple, it is less likely that the triple will be useful. After all, general-purpose architectures are designed to implement almost all of our RTL operators, and many of the rest can be implemented by sequences of such simple, ubiquitous instructions as shifts, logical operations, and two's-complement arithmetic (Warren 2003). Implementing a tile may require a long sequence of instructions, but the amount of *reasoning* required is usually small.

Our utility test estimates *the minimum number of algebraic laws that might be applied* to a Hoare triple before our algorithm may find an implementation of a tile. Given a Hoare triple that computes an expression e, our test covers e with fragments of algebraic laws. The more fragments required to cover e, the less likely e is to pass the utility test.

Our utility test uses fragments of laws because when the establishment procedure is given a goal that includes the left-hand side of a law, rule REDUCE APPLICATIONS or PREPARE OPERATOR may split that goal into subgoals, which are formed from fragments of the original law. To be useful, therefore, a Hoare triple need only compute a fragment of the left-hand side of an algebraic law. For example, to cover the overlined and underlined parts of the expression

$$r_1 := \overline{r_2} \vee \underline{(k \gg_l 16) \ll 16},$$

which is computed by the load upper-immediate instruction on the PowerPC, we can use the overlined and underlined fragments of the following algebraic laws:

$$\begin{aligned} &\overline{0 \vee} x'_L = x'_L \\ &\underline{((x_L \gg_l n) \ll n)} \vee \mathtt{zx}(\mathtt{lobits}_n(x_L)) = x_L. \end{aligned}$$

The expression can therefore be covered using two algebraic laws.

A triple passes our utility test if the expression in its postcondition can be covered using at most *four* algebraic laws; if a triple requires a larger covering, we discard it. Enough triples pass this test to find complete tilesets for the *x*86, PowerPC, and ARM.

We implement our utility test using the algorithm in Figure 8. Function $estimateLaws$ takes a candidate triple and returns the size of the smallest covering of the expression in the postcondition (lines 1 and 2). We compute the size using the mutually recursive functions $expEstimate$ (lines 4–7) and $coverWithLawFragment$ (lines 9–14). Function $expEstimate$ considers a potential covering using every subexpression e' on the left-hand side of every algebraic law (lines 6–7).

Function $coverWithLawFragment$ tries to cover expression e with another expression e'. If both expressions are applications of the same operator, then we try to cover the subexpressions of e. We may be able to cover a subexpression of e inductively, using the corresponding subexpression of e' (line 12). But we may also be able to cover a subexpression of e with another algebraic law by calling $expEstimate$ recursively (line 13). We try both options and use whichever is cheaper.

9.2 Termination of the search for tiles

By using our utility test to discard Hoare triples that are unlikely to lead to tiles, we bound the size of the search space. The postcondition of every triple can be covered by at most four algebraic laws. There are finitely many algebraic laws, so an expression that can be covered with at most four laws has bounded height, so there are finitely many expressions. The machine descriptions mention finitely many locations, so there are finitely many postconditions. For termination, it therefore suffices to ensure that a new Hoare triple is added to the pool only if no existing Hoare triple can establish the same postcondition.

We actually do slightly better. Sometimes an expression such as $r_1 + x_{\#w}$ can be used in any context where another expression such as $r_1 + 1$ can be used. In these cases, we say that the first expression *subsumes* the second. To exploit subsumption, whenever we have a useful candidate triple that computes an expression e, we consider all the triples in the pool that compute the same expression e, modulo locations and constants, which may differ. We then check whether e subsumes, or is subsumed by, any of the other expressions. Triples whose expressions are subsumed are dropped from the pool.

10. Evaluation

Our work is intended to make it easier to retarget compilers, and we evaluate it accordingly:

1. We show that the components we generate are suitable for use in a production compiler.
2. We show that retargeting a compiler using our algorithm and declarative machine descriptions is easier than retargeting well-known compilers.

It is a side benefit that the very same machine descriptions we use have also been used to help retarget other tools.

10.1 Suitability of generated components for production use

We have generated back ends for our research compiler, Quick C--, targeting the *x*86, PowerPC, and ARM architectures. We have evaluated these back ends for suitability by examining the run times of compiled programs and the run time of our compiler, then comparing them with analogous times from production compilers. Relative to a production compiler, Quick C-- has received very few man-hours of development, so Quick C-- cannot be expected to compile the same benchmark suite that a production compiler can handle,

	*x*86 experiments					PowerPC experiments		
Benchmark	**lcc**	**Hand / lcc**	**Generated / lcc**	**gcc -O0 / lcc**	**gcc -O1 / lcc**	**gcc -O0**	**gcc -O1 / gcc -O0**	**Generated / gcc -O0**
compress-95	43.32 s	0.84	**0.80**	1.01	0.69	107.07 s	0.39	**0.72**
go-95	25.84 s	0.98	**0.88**	1.01	0.61	62.03 s	0.40	**0.73**
vortex-95	55.84 s	1.09	**1.10**	1.05	0.77	144.81 s	0.54	**0.73**
mcf-2006	230.24 s	0.94	**0.92**	0.97	0.90	238.63 s	0.77	**0.81**
bzip2-2006	316.06 s	0.91	**0.90**	1.06	0.68	670.14 s	0.43	**0.60**

Table 9. Running times for benchmarks on *x*86 and PowerPC: On each architecture, we give the time in seconds for a baseline compiler, then normalize all other times to the baseline. The Hand and Generated columns represent Quick C-- with hand-written and automatically generated *x*86 back ends. Benchmarks were compiled without debugging or profiling information, then run on an AMD Athlon MP 2800+ with 2 GB of memory and a PowerPC 7447A clocked at 1.25 GHz with 1 GB of memory.

and it cannot be expected to produce equally optimized code. Still, our measurements provide a basis for judging whether our back ends are suitable for production use.

Our choice of benchmarks is limited by the availability of retargetable *front* ends. In order to avoid porting a run-time system to Quick C--, we work only with C benchmarks. Among C compilers, the only one we know of that can cleanly and easily be connected to a new back end is `lcc` (Fraser and Hanson 1995). Unfortunately, because `lcc` is limited to ANSI C, and because Quick C-- does not support `<stdarg.h>`,[4] the combination of `lcc` and Quick C-- can compile only a fraction of typical benchmarks. From the SPEC CPU95 and SPEC CPU2006 benchmark suites, we have chosen those benchmarks that can be compiled by this combination. For each benchmark, we measured the wall-clock running time, averaged over five runs. The CPU95 benchmarks show results on the reference inputs, and the CPU2006 benchmarks show results on the training inputs.

At present, Quick C-- has another significant limitation: it cannot use software libraries to implement RTL operators that are not supported in hardware. Although it would be possible to write algebraic laws to help our algorithm find implementations of unsupported operators, that approach would be impractical: it would lead to bloated executables. The standard technique is to implement a missing operator by making a function call into one of the software libraries that are provided by the manufacturers of the ARM. Because Quick C-- does not yet support this technique, we have run no benchmarks on the ARM. We have, however, verified that our automatically generated ARM back end correctly compiles the integer-only programs in the Quick C-- and `lcc` test suites.

On the *x*86, we compare benchmarks compiled using `lcc`, a hand-written back end for Quick C--, our automatically generated back end, and gcc using both `-O0` and `-O1`. On the PowerPC, no `lcc` back end or hand-written C-- back end is available, so we compare only our automatically generated back end and gcc. Table 9 shows the running times of the benchmarks. Running times of code from baseline compilers are in seconds; other times are given as ratios. Ratios for our automatically generated back ends are highlighted.

Programs compiled by our automatically generated back end run at least as fast as when compiled by the hand-written back end. The one exception is the vortex-95 benchmark, which when compiled by our automatically generated back end is just 0.3% slower. Programs compiled by our automatically generated back end are also comparable to unoptimized code produced by `lcc` or by gcc `-O0`. Although none of the compilers perform many optimizations, code from Quick C-- is often a bit faster, probably because of our peephole optimizer. But when we run gcc with a suite of scalar optimizations (`gcc -O1`), it produces code that is 2% to 30% faster than code produced by our automatically generated back end.

Until we have implemented a substantial set of optimizations in Quick C--, we cannot provide direct, incontrovertible evidence that we can generate a back end for an optimizing compiler that is competitive with gcc. But our results, together with Benitez and Davidson's (1994) prior work showing that standard scalar and loop optimizations can be implemented in a compiler very similar to Quick C--, indicate that our code expanders and recognizers could be used in a high-quality optimizing compiler.

Compiled code should run fast, but the compiler itself should run reasonably fast, too. We measured the time required to compile the Quick C-- test suite using generated and hand-written tilesets (Table 10). When using the automatically generated tileset, compile time increases by about 10%. The extra time is spent not in the code expander but in the optimizer; the compiler takes longer because the generated code expander emits worse code.

10.2 Ease of retargeting

It is easy to measure speed of compiled programs. It is harder to evaluate whether we have developed a good approach to retargeting compilers. In this paper, we evaluate only components related to instruction selection. Machine-dependent components that solve other important problems—register allocation, stack-frame layout, and calling conventions—cannot be generated from declarative machine descriptions alone, even in principle; some knowledge of software convention is required. For that reason, these components are best retargeted using other techniques, as we have described elsewhere (Smith, Ramsey, and Holloway 2004; Lindig and Ramsey 2004; Olinsky, Lindig, and Ramsey 2006).

We compare retargeting effort with two very different compilers: gcc and `lcc`; Dias (2008, §8.2) presents a more thorough case study which also includes vpo. Like Quick C--, gcc is an optimizing compiler that uses a code expander and a recognizer, and like Quick C--, gcc tries to use the full instruction set of the target machine. Although gcc is widely retargeted, it is not clear how much design effort has been invested in making retargeting easy. By contrast, `lcc` has been designed for retargeting and for compilation speed, but not for optimization. It translates low-level intermediate code to assembly code using bottom-up rewriting, which is specified using a BURG mapping from `lcc`'s intermediate code to assembly-code strings (Fraser, Henry, and Proebsting 1992).

All three compilers use domain-specific languages to help create instruction selectors, but the languages are processed very differently. We illustrate the descriptions by using the *x*86 subtract instruction as an example. Quick C-- uses this concise λ-RTL description:

```
SUB^mr^od (Eaddr, reg) is
    Eaddr      := sub (Eaddr, reg)
  | Reg.EFLAGS := x86_subflags(Eaddr, reg)
```

[4] Definition of C-style variadic functions is incompatible with proper tail calls, which every C-- compiler is required to support.

λ-RTL can be used for multiple purposes, but as shown in this paper, creating an instruction selector requires substantial analysis.

The `lcc` compiler uses an even more concise description, which maps a subtract node in `lcc`'s intermediate form to an assembly-language template:

```
reg: SUBI4(reg,mrc) "?movl %0,%c\nsubl %1,%c\n" 1
```

In the template, `%c` represents the target register; `%0` and `%1` represent the two arguments. The two-instruction sequence simulates `lcc`'s three-address node on the two-address target machine.

The `lcc` BURG pattern is very concise, and processing is simple: the compiler finds a minimum-cost covering of each intermediate code, then instantiates the templates. But instantiation is less simple than it may appear. For example, the initial question mark signals a special case in which `lcc` avoids emitting the `movl` instruction when `%0` and `%c` denote the same register. And the template language includes an "escape clause" by which a template can be instantiated by calling the function `emit2`, which can execute arbitrary C code.

Gcc's description of an *x*86 subtract is dramatically different from either a λ-RTL machine description or a BURG mapping:

```
 (define_expand "subsi3"
  [(parallel
   [(set (match_operand:SI 0 "nonimmediate_operand" "")
      (minus:SI
        (match_operand:SI 1 "nonimmediate_operand" "")
        (match_operand:SI 2 "general_operand" "")))
    (clobber (reg:CC FLAGS_REG))])] ""
  "ix86_expand_binary_operator(MINUS,SImode,operands);
   DONE;")

 (define_insn "*subsi_3"
  [(set (reg FLAGS_REG)
     (compare (match_operand:SI 1
                 "nonimmediate_operand" "0,0")
       (match_operand:SI 2 "general_operand" "ri,rm")))
   (set (match_operand:SI 0 "nonimmediate_operand"
      "=rm,r") (minus:SI (match_dup 1) (match_dup 2)))]
  "ix86_match_ccmode (insn, CCmode)
   && ix86_binary_operator_ok (MINUS, SImode, operands)"
  "sub{l}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu") (set_attr "mode" "SI")])
```

The description language exists only to generate gcc's expander, recognizer, and parts of its optimizer; when the description is processed, different parts are used to generate different components, and arbitrary C code can be spliced in. For example, `define_expand` and `define_insn` help generate gcc's expander and recognizer. And to verify that an RTL is a proper two-address RTL, whose destination is the same as the first operand, the recognizer calls C function `ix86_binary_operator_ok`.

Examining a single instruction illuminates the *kind* of effort required to retarget each compiler: Our work requires only compiler-independent descriptions of instructions' semantics, but it must be supplemented by a compiler-compiler that generates compiler-specific expanders and recognizers. Both `lcc` and gcc use descriptions organized around their respective low-level intermediate codes, supplemented by C code that is both compiler- and target-specific. Reflecting years of design effort, `lcc`'s descriptions are an order of magnitude simpler than gcc's.

We now turn to the *degree* of effort required to retarget each compiler. How many instructions does each compiler exploit, and at the cost of what effort? We believe that number of instructions recognized predicts the code quality achievable with a back end, because the more instructions the recognizer accepts, the more code-improving transformations an optimizer can apply.

Back end	Expansion Time (s)	Compile Time (s)
Generated	0.87	23.72
Hand-written	0.77	21.45

Table 10. Compile times for the Quick C-- test suite (10,718 lines in 91 files) using generated and hand-written *x*86 back ends. Times are averaged over five runs, with each run compiling the entire test suite to assembly code with one invocation of the compiler. The compiler ran on a Pentium M, 1.5 GHz with 1 GB of memory.

As expected, `lcc` exploits the fewest instructions: a code generator based on BURG covers the compiler's intermediate code, not the target instruction set. In `lcc` 4.2, the back end uses 249 BURG patterns but only 53 instruction opcodes; the "escape hatch" is used in 8 patterns but uses only one additional opcode. The back end uses 549 lines of BURG description plus 463 lines of target-dependent C code.

The compilers designed for optimization require larger descriptions and more target-dependent code, but they exploit many more instructions. In the hand-written Quick C-- back end for the *x*86, the tileset and recognizer require 1,286 lines of code but can generate only 233 distinct instructions. In the automatically generated Quick C-- back end, the λ-RTL and SLED descriptions require 1,948 lines, but they describe 639 distinct instructions. (And the SLED description has been reused to create other tools, including a binary translator, a link-time optimizer, and a dynamic code generator, so perhaps not all of its lines should be charged only to Quick C--.) In gcc 4.2.1, the compiler-specific "machine description" requires 18,462 lines to exploit 754 instructions and to describe 320 expansions and 83 machine-specific peephole optimizations.

Among the four back ends, `lcc` may have the most impressive power-to-weight ratio, but our automatically generated instruction selectors are competitive, and they support code-improving transformations, such as peephole optimization, that BURG does not support. And when we compare the two automatically generated optimizing compilers, the contrast between declarative machine descriptions and gcc's "machine descriptions" cannot be overstated. Gcc's machine description is an order of magnitude longer than the λ-RTL machine description, suggesting that retargeting effort will be proportionally greater. There are also significant differences in kind. Declarative descriptions can be tested independently (Fernández and Ramsey 1997; Bailey and Davidson 2003); gcc's descriptions cannot. The declarative machine description concisely describes a property of a machine, independent of any particular tool, whereas gcc's domain-specific code is verbose and requires knowledge of gcc's internals. More significantly, gcc's retargeting strategy relies on textual inclusion of arbitrary snippets of C code, including calls to hand-written, target-specific functions. This strategy makes it unlikely that one of gcc's machine descriptions could be analyzed or reused for another purpose. Finally, we believe that the compiler writer's task is made more difficult by gcc's tactic of writing critical code fragments by hand and using a macro framework to include those fragments in the right parts of the generated back end.

11. Related Work

People have been working on retargetable compilers since 1958 (e.g., Strong et al.). We discuss only the most closely related work: back ends or optimizers generated using machine descriptions or search techniques. We focus on the search strategy used to find instructions and the pruning heuristic used to guarantee termination.

The most closely related work was conducted thirty years ago by Cattell (1980). Cattell pioneered some ideas that we use, including rewriting with algebraic laws and compensating for unwanted assignments by using temporaries and scratch registers. But unlike our search, Cattell's search starts with the goal of implementing a particular computation (analogous to one of our expansion tiles) and searches for an implementation composed of machine instructions. Given a goal, Cattell's algorithm rewrites the goal using an algebraic law. A heuristic chooses which law to apply; the heuristic looks for a law that it estimates will produce a new goal that is somehow closer to machine instructions. Cattell prunes the search by limiting both the depth of rewriting and the number of laws which may be used to rewrite a single expression. These limits guarantee termination, but they place a tremendous burden on the heuristic that chooses laws: it must prioritize the algebraic laws that are likely to lead to machine instructions. By contrast, our search does not limit the depth of rewrites *a priori*, and it attempts to apply every applicable law.[5]

Hoover and Zadeck (1996) present the `TOAST` machine-description language and an algorithm for generating instruction selectors. Their algorithm tries to recursively match a goal expression with the expression computed by an assignment in a machine instruction. They prune the search by checking the result of each rewrite: search continues only if the root of the resulting goal expression matches the expression in the machine instruction. This heuristic ensures that the search is bounded by the size of the expression computed by the machine instruction. But this pruning heuristic misses opportunities for one algebraic law to rewrite the results produced by applying another algebraic law.

Ceng et al. (2005) generate most of a BURG-style instruction selector using a `LISA` machine description. They have two ways to find an implementation of their equivalent of a tile: there may be a machine instruction with an assignment that implements the tile; or if not, they have a collection of rewrite rules, one of which may implement the goal tile using other tiles. Their search does not find implementations of every tile; some are implemented by hand. Unlike many others, Ceng et al. present experimental results: code produced by their generated back end is about 5% slower than code produced by a hand-written back end. The instruction selector runs after the optimizer, so when the instruction selector produces inefficient code, the final program is also inefficient.

Remarkably, it is possible to retarget a back end without a machine description, by extracting knowledge from an existing C compiler (Collberg 1997). Collberg's algorithm generates hundreds of small, carefully crafted sample programs, compiles them, and analyzes the resulting assembly code. Collberg defines a simple, RISC-like intermediate code, and his algorithm uses the results of the analyses to generate a mapping from that code to target instructions. The mapping is given to BEG (Emmelmann, Schröer, and Landwehr 1989), which generates a back end. Although Collberg's techniques are largely unrelated to the other techniques discussed here, his techniques are ingenious, the degree of automation is impressive, and the work repays careful study.

A less closely related problem is *superoptimization*: a search for more efficient ways of implementing computations whose implementations on a target machine are already known (Massalin 1987; Granlund and Kenner 1992). Massalin's superoptimizer exhaustively combines sequences of machine instructions, then executes each sequence against test inputs to see if the sequence implements a goal computation. The superoptimizer doesn't use a machine description; it uses the machine itself. The search is pruned by rejecting sequences, either because a subsequence is known to be suboptimal or because a later instruction in the sequence does not use the results of any earlier instructions. This pruning does not guarantee termination, but since the goal of a superoptimizer is to find ways of making a working back end better, a superoptimizer needn't be guaranteed to terminate—it can be killed at any time.

Joshi, Nelson, and Zhou (2006) present the Denali superoptimizer, which does use a machine description. The description is declarative, and it takes the form of axioms given to a refutation-based theorem prover. Denali uses goal-directed search to find a sequence of machine instructions that implements a goal computation. The search is limited by a *matcher*, which enumerates equivalence relations for each expression in the input code. (It is not clear how Denali ensures that each equivalence class is finite; the published examples suggest that the technique involves limiting the set of algebraic laws.) After identifying equivalent ways to implement the goal computation, Denali uses a SAT solver to search for implementations that use the fewest processor cycles. Denali finds high-quality implementations of register-to-register computations.

12. Conclusion

In this paper, we generate an instruction selector using *declarative* machine descriptions, which give only the syntax and semantics of target-machine instructions. We show that the problem is undecidable, so a heuristic search is indicated. We present a new search technique and a new pruning heuristic, which rely on three key ideas:

- We search only computations that we know can be implemented entirely by machine instructions.
- When a sequence of instructions implements an expansion tile, we can discover this sequence by matching the left-hand side of an algebraic law with the expression computed by the final instruction in the sequence.
- When we match two expressions, we try unification, but when the expressions involve machine state, we instead search for a sequence of machine instructions whose execution makes the expressions equal.

Our technique finds not just implementations but also valid Hoare triples, which could be used to generate a certified code expander (cf Leroy 2006).

Our new search algorithm generates high-quality instruction selectors for the *x*86, PowerPC, and ARM. And thanks to Davidson and Fraser's (1980) compiler design, our back ends, unlike earlier automatically generated back ends, produce code that is as good as the code produced by hand-written back ends.

Acknowledgments

This work was funded by a grant from Intel Corporation and by NSF awards 0838899 and 0311482. We are grateful for extensive feedback from anonymous reviewers, especially those who pressed for deeper treatment of our results.

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.

[5] To be fair to Cattell, technology has changed dramatically, and our algorithm might be unworkable on 1980 hardware.

Joel Auslander, Matthai Philipose, Craig Chambers, Susan Eggers, and Brian Bershad. 1996 (May). Fast, effective dynamic compilation. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 31(5):149–159.

Mark W. Bailey and Jack W. Davidson. 2003 (November). Automatic detection and diagnosis of faults in generated code for procedure calls. *IEEE Transactions on Software Engineering*, 29(11):1031–1042.

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.

Mark Bezem, Jan Willem Klop, and Roel de Vrijer, editors. 2003. *Term Rewriting Systems*. Cambridge University Press, Cambridge, UK.

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.

Jianjiang Ceng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. 2005. C compiler retargeting based on instruction semantics models. In *Design, Automation, and Test in Europe*, pages 1150–1155. IEEE Computer Society.

Cristina Cifuentes, Mike Van Emmerik, and Norman Ramsey. 1999 (October). The design of a resourceable and retargetable binary translator. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE'99)*, pages 280–291.

Christian S. Collberg. 1997 (May). Reverse interpretation + mutation analysis = automatic retargeting. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 32(5):57–70.

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

Jack W. Davidson. 2008. Personal communication, 14 November 2008.

Jack W. Davidson and Christopher W. Fraser. 1980 (April). The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202.

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. 2008 (December). *Automatically Generating the Back End of a Compiler Using Declarative Machine Descriptions*. PhD thesis, Harvard University, School of Engineering and Applied Sciences. As of July 2009, at `http://tinyurl.com/lxhly5`.

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.

Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.

Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. 1989 (July). BEG—a generator for efficient back ends. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 24(7):227–237.

Lee D. Feigenbaum. 2001 (April). Automated translation: generating a code generator. Technical Report TR-12-01, Computer Science, Harvard University.

Mary F. Fernández and Norman Ramsey. 1997 (May). Automatic checking of instruction specifications. In *Proceedings of the International Conference on Software Engineering*, pages 326–336.

Mary F. Fernández. 1995 (June). Simple and effective link-time optimization of Modula-3 programs. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 30(6):103–115.

Christopher W. Fraser. 1989 (July). A language for writing code generators. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 24(7):238–245.

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

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.

Torbjoern Granlund and Richard Kenner. 1992. Eliminating branches using a superoptimizer and the GNU C compiler. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 27(7):341–352.

Roger Hoover and Kenneth Zadeck. 1996. Generating machine specific optimizing compilers. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 219–229.

Rajeev Joshi, Greg Nelson, and Yunhong Zhou. 2006. Denali: A practical algorithm for generating optimal code. *ACM Transactions on Programming Languages and Systems,* 28(6):967–989.

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Conference Record of the 33rd Annual ACM Symposium on Principles of Programming Languages*, pages 42–54.

Christian Lindig and Norman Ramsey. 2004 (April). Declarative composition of stack frames. In *13th International Conference on Compiler Construction (CC 2004), LNCS* volume 2985, pages 298–312.

Henry Massalin. 1987 (October). Superoptimizer: A look at the smallest program. *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS II),* in *SIGPLAN Notices*, 22(10):122–127.

Reuben Olinsky, Christian Lindig, and Norman Ramsey. 2006 (January). Staged allocation: A compositional technique for specifying and implementing procedure calling conventions. In *Proceedings of the 33rd ACM Symposium on the Principles of Programming Languages*, pages 409–421.

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.

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 and Mary F. Fernández. 1995 (January). The New Jersey Machine-Code Toolkit. In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302.

Norman Ramsey and Mary F. Fernández. 1997 (May). Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524.

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.

Michael D. Smith, Norman Ramsey, and Glenn Holloway. 2004 (June). A generalized algorithm for graph-coloring register allocation. *ACM SIGPLAN '04 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 39(6):277–288.

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.