To appear in the *15th International Conference on Compiler Construction*

Converting Intermediate Code to Assembly Code Using Declarative Machine Descriptions

João Dias and Norman Ramsey

Division of Engineering and Applied Sciences
Harvard University

Abstract. Writing an optimizing back end is expensive, in part because it requires mastery of both a target machine and a compiler's internals. We separate these concerns by isolating target-machine knowledge in *declarative machine descriptions*. We then analyze these descriptions to automatically generate machine-specific components of the back end. In this work, we generate a *recognizer*; this component, which identifies register transfers that correspond to target-machine instructions, plays a key role in instruction selection in such compilers as vpo, gcc and Quick C--. We present analyses and transformations that address the major challenge in generating a recognizer: accounting for compile-time abstractions not present in a machine description, including variables, pseudo-registers, stack slots, and labels.

1 Introduction

Because of the substantial effort required to build a compiler, and because of the increasing diversity of target machines, including PCs, graphics cards, wireless sensors, and other embedded devices, a compiler is most valuable if it is easily retargeted. But in current practice, retargeting requires new machine-dependent components for each new back end: instruction selection, register allocation, calling conventions, and possibly machine-specific optimizations. Writing these components by hand requires too much effort, and it requires an expert who knows both the internals of the compiler and the details of the target machine. Our long-term goal is to minimize this effort by dividing and encapsulating expertise: compiler expertise will be encapsulated in compiler-specific optimizations and code-generator generators, and machine expertise will be encapsulated in *declarative machine descriptions*.

A declarative machine description clearly and precisely describes a property of a machine, in a way that is independent of any compiler. For example, the SLED machine-description language (Ramsey and Fernández 1997) describes the binary and assembly encodings of machine instructions, and the λ-RTL machine-description language (Ramsey and Davidson 1998) describes the semantics of machine instructions. A declarative machine description is well suited to formal analysis, and because it is independent of any compiler, it can be reused by multiple compilers and other tools. Furthermore, a declarative machine description may be checked independently for correctness or consistency (Fernández and Ramsey 1997).

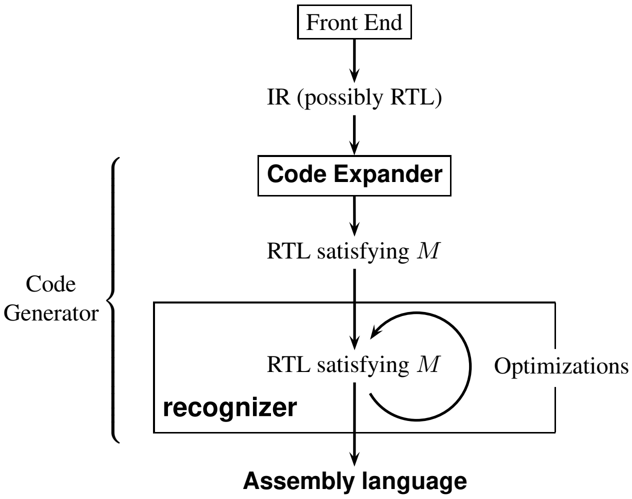

Fig. 1. Davidson/Fraser compiler: M represents the machine invariant. Machine-dependent components and representations are in **bold**.

Our ultimate goal is to use declarative machine descriptions to generate *all* the machine-dependent components of a compiler's back end. In this work, we generate a *recognizer*, which supports machine-independent instruction selection and optimization (Davidson and Fraser 1984). A recognizer is an integral part of Davidson and Fraser's compilation strategy: intermediate code is represented by machine-independent *register-transfer lists* (RTLs), but each RTL is required to be implementable by a single instruction on the target machine (Figure 1). This requirement, called the *machine invariant*, is established by a compiler phase called the *code expander*, which runs at the start of code generation. In later phases, the requirement is enforced by the recognizer: each phase is built around a machine-independent, semantics-preserving transformation, and the phase maintains the machine invariant by asking the recognizer if each new RTL can be implemented by an instruction on the target machine; if not, transformations are rolled back. For example, instead of building a peephole optimizer for each target, we build a single peephole optimizer which combines related instructions and, if the recognizer accepts the combination, replaces the original instructions with the combination. The recognizer can not only identify which RTLs correspond to machine instructions but can also emit assembly code for such instructions.

In generating a recognizer from machine descriptions, the major challenge is to account for compile-time abstractions such as variables, pseudo-registers, stack slots, and labels. Such abstractions, while essential to compilation, have no place in a machine description. The contribution of this paper is a set of analyses and transformations that enable us to bridge this "semantic gap" between instructions as viewed by a machine and instructions as viewed by a compiler. We have built these analyses and transformations into a "λ-RTL toolkit," which generates recognizers for our Quick C-- compiler.

2 The semantic gap

The λ-RTL machine-description language takes the perspective of the bare machine. Each instruction is specified as a transformation on the machine's state. This state is modeled as a collection of *storage spaces*, each of which is an array of *cells*. For example, on the x86, the storage spaces include the 'r' space for the 32-bit integer registers, the 'm' space for 8-bit addressable memory, and the 'f' space for the floating-point register stack. We refer to storage using array-index notation; for example, `$r[0]` refers to the first cell in the 'r' space.

Transformations on storage are specified using a formal notation for register transfers (Ramsey and Davidson 1998). For example, `$r[0] := 16 + $r[0]` adds 16 to the value in register 0, then places the sum in register 0. The register transfers needed to describe a machine are so simple that this example shows essentially all their elements: storage, assignment, literal bit vectors such as 16, and RTL operators such as $+$.

A compiler has a much richer model of computation. During compilation, computations may refer to source-language variables, pseudo-registers, stack slots whose locations have not yet been determined, and names defined in separately compiled modules. A compiler also distinguishes among storage spaces in ways that are not necessary in a machine description; for example, hardware registers are typically managed entirely by the compiler, whereas memory locations are managed partly by the compiler (e.g., stack slots) and partly by user code (e.g., heap). A compiler therefore needs a much richer model of register transfers than a machine-description language:

- Hardware locations are not undifferentiated storage cells; a compiler represents registers differently from memory. Moreover, compile-time locations include not only hardware locations but also variables and pseudo-registers.
- Constants include not only literal bit vectors but also late compile-time constants (e.g., the offset of a stack slot) and labels.

The differences in the representations of locations and constants constitute the semantic gap between λ-RTL's model of the machine and a compiler's model of the machine.

2.1 Mapping high-level RTLs to low-level RTLs

To be usable at any time during compilation, a recognizer must accept RTLs containing high-level abstractions such as variables, labels,[1] stack slots, and late compile-time constants. To accept such a high-level RTL, the recognizer must know how the compiler will map that RTL down to a machine-level RTL. That way, it can accept an RTL if and only if the RTL's image (under the mapping) will be implementable by a single instruction on the target machine.

The mapping is distributed over several compiler phases, each of which eliminates one abstraction. Variables and labels are familiar, simple abstractions, but stack slots and late compile-time constants may need some explanation.

[1] Our compiler actually works with "link-time constant expressions," which include not only labels but also such expressions as the sum of a label and a constant or the difference of two labels. But for simplicity, in this paper we refer to all of these expressions as "labels."

We represent a stack slot as a memory location addressed using an offset from a frame pointer. Until stack-frame layout is complete, we represent the offset as a symbolic constant (Lindig and Ramsey 2004); such a constant is called a *late compile-time constant* and is notated k. Furthermore, in order to save a register, our compiler uses a *virtual* frame pointer (vfp), which is read-only. At each program point, the virtual frame pointer is at a known offset from the stack pointer, but because the stack pointer may move, offsets may differ at different program points. After stack layout is complete, these offsets are computed by a separate dataflow pass, which also replaces each use of the virtual frame pointer with an expression of the form stack-pointer-plus-constant.

Figure 2 shows all phases of our back end; code flows from top to bottom. Phases that map high-level RTLs to low-level RTLs are interleaved with other phases, such as the code expander and the optimizer. The first phase, the variable placer, puts each variable in a pseudo-register or a stack slot. Then the code expander establishes the machine invariant: provided that suitable substitutions are made for pseudo-registers, the virtual frame pointer, late compile-time constants, and labels, each RTL can be represented by a single instruction on the target machine. The optimizer then improves the code, maintaining this form of the machine invariant. After optimization, the register allocator replaces pseudo-registers with hardware registers; the stack freezer lays out the stack frame and replaces late compile-time constants with constant bit vectors; and a final pass replaces each use of the virtual frame pointer with the correct stack-pointer-plus-offset expression. The only remaining high-level abstractions are labels, which are dealt with by the assembler.

The phases of our compiler behave as in typical compilers, with two exceptions: the expander generates very naïve code, and the optimizer works at the machine level. Generating naïve code makes it easier for the optimizer to find redundancies. And exposing the machine-level semantics of the target instructions makes it possible for the optimizer to produce better code (Benitez and Davidson 1994).

To generate a recognizer for the high-level RTLs manipulated by the compiler, we use our λ-RTL toolkit to analyze the low-level RTLs in a machine description and find the corresponding set of high-level RTLs. Specifically, our analyses recover high-level abstractions of locations and constants from the low-level RTLs in the λ-RTL machine description. The basic idea behind the analyses is to invert the compiler's mapping from high-level RTLs to low-level RTLs. Some analyses are very nearly the inverse of compiler stages; others are not. Before presenting these analyses in Section 3, we explain how the semantics and encoding of instructions are expressed in the λ-RTL and SLED machine-description languages.

2.2 Describing instructions using λ-RTL and SLED

λ-RTL and SLED share the same model of an instruction set: a simple grammar gives the abstract syntax of instructions and addressing modes. For example, the x86 8-bit add-immediate instruction is associated with the abstract syntax `ADDidb (reg, i8)`, where `ADDidb` is a *constructor* and `reg` and `i8` are integer *operands* (a register number and an 8-bit immediate operand, respectively). A constructor acts much like an opcode, but although opcodes may be overloaded in the surface syntax of an assembly language, constructor names are not overloaded; the `idb` suffix serves to distinguish this instruction from other add instructions.

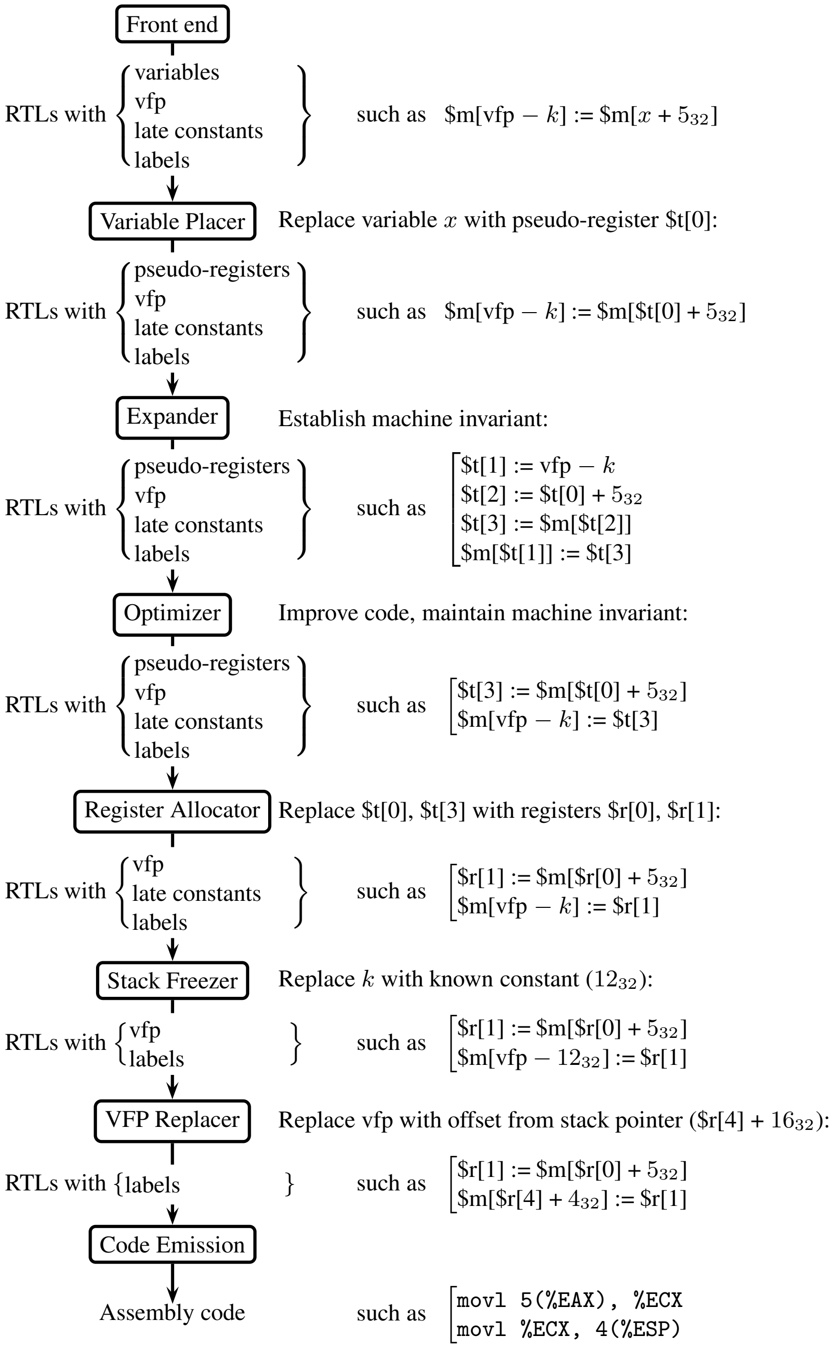

Fig. 2. Translation by phases. Each phase is shown in a box; each arrow between phases describes the representation passed between those phases. The examples on the right show the evolution of a memory-to-memory move instruction.

Using this model, a λ-RTL description associates each abstract-syntax tree with a semantics (Ramsey and Davidson 1998). More precisely, each constructor is associated with a function that maps the semantics of the operands to the semantics of the instruction, which is described using a low-level RTL. For example, neglecting effects on condition codes (Section 4), the semantics of `ADDidb` is

```
ADDidb (reg, i8) is $r[reg] := $r[reg] + sx i8
```

The `ADDidb` instruction stores the sum of register `$r[reg]` and the sign-extended 8-bit immediate `i8` back into register `$r[reg]`.

In similar fashion, a SLED description associates each instruction with an assembly-language and binary representation (Ramsey and Fernández 1997). The details are beyond the scope of this paper.

Given λ-RTL and SLED machine descriptions, we generate a recognizer that matches a high-level, compile-time RTL with an instruction represented using the common abstract syntax. From this representation, we produce assembly or binary code.

3 Transforming λ-RTL into a recognizer

We generate a recognizer by transforming a λ-RTL description into an automaton that can accept high-level RTLs. Broadly speaking, this transformation involves two tasks: bridging the semantic gap and working within the limitations of efficient automata.

To bridge the semantic gap, we transform the low-level RTLs in the machine description into patterns that match the high-level RTLs used in our compiler. In particular, we arrange to accept registers and memory (in place of undifferentiated storage spaces), pseudo-registers (in addition to hardware registers), stack slots, late compile-time constants, and labels. Accepting each of these high-level abstractions requires some analysis or transformation of the original λ-RTL.

The primary limitation of efficient automata is that it is not known how to accept or reject an RTL efficiently based on its semantics; we can do so only based on its syntax. This limitation affects the compiler: a transformation maintains the machine invariant only if the recognizer accepts the transformed RTLs. To manage this limitation requires both an additional compile-time invariant and additional analysis and transformation of the original λ-RTL.

We achieve these two tasks through a combination of different techniques, as detailed below. The whole is a bit of a bag of tricks, but there is one pleasantly recurring theme: binding time.

3.1 Bridging the semantic gap

We cover the high-level abstractions from the simplest to the most complex: labels, registers and memory, pseudo-registers, and stack slots.

Labels Labels are easy because they are supported by the assembler and linker, which deal with the semantic gap. Coding of labels within machine instructions is the province of SLED, which handles not only labels coded as absolute addresses but also labels coded using PC-relative arithmetic. By hiding this coding, SLED simplifies the λ-RTL description and thereby our recognizer. For example, λ-RTL describes an x86 PC-relative jump instruction as follows:

```
JMP.Jb (addr) is EIP := addr
```

The jump instruction takes an address, which could be a label, and sets the program counter `EIP` to the value of the label. Because the SLED description identifies which operands can be labels, our toolkit need only identify `addr` as a possible label and generate code to match it.

Registers and memory The first part of the semantic gap that requires work on our part is the classification of each storage space as registers or memory. We use a *binding-time analysis* developed by Feigenbaum (2001). Depending on when the value of an expression becomes known, Feigenbaum identifies three binding times:

- *Specification time:* The value of the expression depends only on literal constants, so it may be determined from the λ-RTL specification alone.
- *Instruction-creation time:* The value of the expression depends only on the values of an instruction's operands, so it may be determined when the instruction is created.
- *Run time:* The value of the expression depends on machine state, so it is not determined until run time.

A simple analysis determines the binding time of each expression in a machine description. To distinguish registers from memory, Feigenbaum applies this analysis to the addressing expressions used to compute cell numbers in each storage space. The resulting binding times classify the spaces:

- *Fixed space:* The value of each addressing expression is determined by the constructor used to build the instruction, so these values are known at specification time. An example on the x86 is the control-register space, which contains the program counter and the condition codes.
- *Register-like space:* The value of each addressing expression is determined by constructors and operands, so these values are known at instruction-creation time. An example on the x86 is the integer-register space.
- *Memory-like space:* The values of some addressing expressions may not be known until run time. An example on the x86 is the memory space. A more subtle example is the x86 floating-point register stack: it may be indexed using the run-time value of the floating-point stack-top pointer.

The λ-RTL toolkit classifies fixed and register-like spaces as registers; it classifies memory-like spaces as memory. It then transforms the RTLs in the machine description to distinguish between registers and memory, just like the RTLs manipulated by the compiler.

By choosing the operands used in addressing expressions in a register-like space, a compiler can control which cells are used. The compiler can therefore manage these cells using standard register-allocation techniques, justifying the name “register-like.” The next step in our transformation is therefore to arrange for the recognizer to accept pseudo-registers, which will be mapped to hardware registers by the register allocator.

Pseudo-registers Each pseudo-register lives in an imaginary, infinite storage space that is different from any hardware space. Pseudo-register spaces are not one-to-one with hardware register spaces; instead, each pseudo-register space corresponds to an *interchangeable set* of hardware registers. Such a set is not necessarily identified with a hardware register space; for example, on the SPARC, integer registers $r[1] to $r[31] are interchangeable in most instructions, but integer register $r[0] is not, because it is hardwired to zero. As another example, a single hardware space may include multiple, distinct register sets; on the x86, for example, the 32-bit, 16-bit, and 8-bit integer registers each form distinct sets, despite the fact that the 16-bit registers are contained entirely within the 32-bit registers, and the 8-bit registers are contained entirely within the 16-bit registers. Our λ-RTL toolkit identifies these register sets, associates each set with a pseudo-register space, and arranges for the recognizer to accept a pseudo-register if and only if it would accept *any* register from the corresponding set.

To identify sets, we use a *location-set analysis* developed by Feigenbaum (2001):

– *Fixed location set:* A fixed location set contains only a single location. A fixed location set arises if an instruction refers specifically to a location. For example, the 32-bit multiply instruction on the x86 refers to the `EAX` register; no other location could replace `EAX` in this instruction. The compiler simply uses the location; no pseudo-registers are needed.
– *Register-like location set:* A register-like location set contains a set of locations from a register-like space. For each register-like location set, the toolkit introduces a new pseudo-register space.
– *Memory-like location set:* A memory-like location set contains a set of locations from a memory-like space. For example, a load from memory on the x86 may load from any memory location.

The analysis works by examining the addressing expression in each RTL location in each instruction. If the addressing expression is always bound at specification time, the location forms a fixed location set. If the addressing expression is bound at instruction-creation time or at run time, the analysis assumes it may evaluate to any bit vector of the appropriate width, except that the value may be constrained by guards on the RTL containing the location. The location set consists of those cells whose numbers satisfy the constraints. For example, on the SPARC, most instructions are guarded by a condition specifying that the addressing expression for an integer register is nonzero, and the relevant location set contains only integer registers $r[1] to $r[31].

After the location-set analysis, the λ-RTL toolkit replaces each location in an RTL with a pattern that matches either a hardware location or an appropriate pseudo-register. The compiler and the λ-RTL toolkit must agree on the names used to represent pseudo-register spaces.

Stack slots and late compile-time constants For most of compilation, a stack slot is a memory reference of the form \$m[vfp + k], where k is a (symbolic) late compile-time constant. Because the address in this form must be accepted wherever the hardware would expect a reference of the form stack-pointer-plus-constant, we must extend the recognizer to deal with both the virtual frame pointer and late compile-time constants.

The virtual frame pointer is eventually replaced with an expression of the form $\mathrm{sp} + n$, where sp is the stack pointer and n is a literal bit vector. We therefore arrange to accept the virtual frame pointer wherever $\mathrm{sp}{+}n$ would be accepted. The only potentially tricky part is identifying the stack pointer. If there is only one indistinguishable set of registers used in addressing expressions, we can simply assume the stack pointer is in that set. Otherwise, because the identity of the stack pointer is a matter of software convention, the λ-RTL toolkit must be told which register is the stack pointer.

The more difficult problem is when to accept a late compile-time constant k. The problem is that an instruction set may limit the number of bits of precision available for an immediate constant. For example, while the x86 supports 32-bit immediate constants, the MIPS supports only 16-bit constants, and the SPARC only 13-bit constants. The recognizer can easily determine if the value of a literal bit vector fits in 16 or 13 bits, but what should it do with a 32-bit late compile-time constant, which is symbolic?

One solution is to be pessimistic: to reject any late compile-time constant that might be too wide. This solution requires that the code expander be pessimistic as well. For example, to address a memory cell using the expression $\mathrm{sp}{+}k$, the code expander might load the high bits of k into a pseudo-register, add sp to that pseudo-register, and then address the cell by using the low bits of k as an offset from the pseudo-register. After the stack-freezing phase determines the values of late compile-time constants, some of these extra instructions might be eliminated by the peephole optimizer, but in the meantime the compiler must deal with more instructions in the intermediate representation, as well as increased register pressure.

A better solution is to assume optimistically that a late compile-time constant fits in the width required by the instruction. This solution results in a simpler code expander, a simpler recognizer, and fewer instructions in the intermediate representation. It works well because most late compile-time constants represent offsets of stack slots, which are usually small. But when the optimistic assumption proves incorrect, the compiler must fix any incorrect code. On some machines, the compiler must reserve a register for such fixup code. Sometimes the assembler will reserve a register and do the fixup, allowing compiler writers to assume that machine instructions can handle any 32-bit constant (Kane and Heinrich 1992).

3.2 Limitations of efficient automata

A recognizer should accept any compile-time RTL that is equivalent to some RTL in the machine description. But when are two RTLs equivalent? Ideally, two RTLs would be deemed equivalent if, when executed, they had the same effect on a machine's state. But such equivalence is extremely expensive to compute—and because the recognizer is consulted in the optimizer's inner loop, it has to decide the question efficiently. The need for efficiency rules out reasoning about the effects of RTLs; instead, we decide equivalence based on syntax. It would be pleasant to be flexible and to accept multiple ways of writing such associative-commutative operations as two's-complement addition or simultaneous composition of effects, but even the equivalence relation induced by associativity and commutativity is too expensive to be decided in the inner loop. Accordingly, like vpo and `gcc`, we deem two RTLs to be equivalent only if they are syntactically identical. This impoverished equivalence relation can be decided easily and relatively cheaply at compile time, but to make it useful, we have to work harder at recognizer-generation time.

To illustrate the most frequent way in which a compiler may generate semantically equivalent but syntactically different RTLs, we return to the `ADDidb` instruction. The machine description says

```
ADDidb (reg, i8) is $r[reg] := $r[reg] + sx i8
```

The RTL `$r[0] := $r[0] +` 12_{32} is not a syntactic match for any RTL generated by the right-hand side, but we would like to accept it as a proxy for the semantically equivalent `$r[0] := $r[0] + sx` 12_8, which *is* a syntactic match. We can frame the requirement in terms of binding time: the recognizer should accept a constant in place of an expression that can be evaluated at instruction-creation time. We call such an expression a *compile-time constant expression*.

It is not safe to accept *any* literal constant in place of a compile-time constant expression. For example, the literal constant 65535_{32} could not be obtained by sign-extending an 8-bit immediate constant. In the general case, a literal constant is acceptable only if it satisfies a *constraint*, which ensures that the constant could have been computed by the original expression. For example, for `ADDidb`, the λ-RTL toolkit identifies the compile-time constant expression `sx i8`, and it transforms `sx i8` into a constrained pattern variable `const`:

```
ADDidb (reg, i8) is $r[reg] := $r[reg] + (const : #32 bits)
  where fits_signed(const, 8)
```

By itself, `const` would match any 32-bit constant; the `where` constraint ensures that `const` can be obtained by sign-extending an 8-bit quantity. Using the optimistic strategy described above, the recognizer also allows `const` to match expressions involving only literal bit vectors and late compile-time constants.

The transformation of `sx i8` to `const` has one more subtle consequence. Because SLED uses the original operands of an instruction to construct the assembly or binary encoding of that instruction, the λ-RTL toolkit must generate code to reconstruct the values of those operands. In the `ADDidb` instruction, for example, the value of the operand `i8` can be extracted directly from the value of `const`. The final result is an instruction with constants in place of compile-time constant expressions, with constraints to maintain the original semantics of the instruction, and with a map that can compute the values of the operands of the original instruction:

```
ADDidb (reg, i8) is $r[reg] := $r[reg] + (const : #32 bits)
  where fits_signed(const, 8)
  and   i8 = lobits(const, 8)
```

Finally, once the recognizer is geared to accept constants, the compiler must arrange that *every* compile-time constant expression is represented by a literal constant. In other words, the compiler must fold constants. Constant folding is done by the *simplifier*, which is applied to each RTL before the RTL is passed to the recognizer.

4 Pragmatics

In generating a recognizer, we must deal with two sets of pragmatic concerns: how to manage multiple forms of the machine invariant, and how to manage complexity in the semantics of the target machine.

As they pass through the compiler, RTLs satisfy successively stronger forms of the machine invariant, containing successively fewer high-level abstractions (Figure 2). Because the recognizer is used both in the optimizer and in the code emitter, it must accept RTLs satisfying different forms of the invariant. We could generate multiple recognizers, but it is simpler to generate a single recognizer that accepts the weakest form of the invariant. When used for code emission, this recognizer may mistakenly accept RTLs that contain pseudo-registers or late compile-time constants, but such RTLs can exist only if the register allocator or the stack freezer is broken. If necessary, we could add a function to reject any RTL containing a pseudo-register or late compile-time constant.

Another pragmatic concern is that real machines are often complicated in detail but simple in the abstract. To eliminate unwanted detail, a compiler writer can ignore unused instructions and machine state (e.g., obscure parts of the processor status word).

What about state that is used, but the details of which we wish to ignore? For example, most compiler writers don't care about the x86's six different condition-code bits; they just want to know how conditional-branch instructions interact with instructions that set condition codes. A common trick is to aggregate and abstract over such state. For example, a description of the x86 might treat the condition-code register as an aggregate instead of as six individual bits. Each effect on the aggregate could then be described as the result of some machine-specific comparison operator. For example, to describe the addition instructions, we might introduce the machine-specific operator `x86_addflags`, which takes two n-bit arguments and returns a new value for the entire 32-bit condition-code aggregate:

```
rtlop x86_addflags : #n bits * #n bits -> #32 bits
```

Using this kind of abstraction, the full semantics of `ADDidb` can be described by simultaneous composition of just two effects:

```
ADDidb (reg, i8) is $r[reg] := $r[reg] + sx i8
                  | EFLAGS := x86_addflags($r[reg], sx i8)
```

Judicious use of such abstractions can simplify both a machine description and a compiler, but to ensure that such abstract RTLs are recognized, the machine description and compiler must use exactly the same abstraction to specify the semantics of each instruction.

Using these kinds of abstractions has a number of advantages:

- It is easier to write and understand code that manipulates simpler RTLs.
- The compile-time representation of a simple RTL requires less memory.
- A recognizer that only needs to match simple RTLs may be smaller and faster.

But simplifying abstractions must be used with care; they may change the semantics of instructions in subtle ways. For example, aggregation of mutable state may indicate that an instruction uses or modifies more state than it actually does. It is safe to aggregate mutable state only if no source program can tell the difference between instructions with and without this simplification. Because most source languages do not expose condition codes or status bits, it is usually safe to abstract over mutation of the entire condition-code register.

Even when it is safe, a simplifying abstraction may inhibit optimization. If the optimizer does not know the semantics of each machine-specific operator, it cannot tell when two such operators affect relevant state in the same ways, and it may miss opportunities to remove redundant code.

5 Generating a recognizer

To generate a recognizer, we first use analyses from Section 3 to transform the λ-RTL description into a suitable pattern match over compiler RTLs. This match is expressed in terms of a one-off, domain-specific language. This language defines a set of nonterminals, each of which may be matched by BURG-style, linear tree patterns which are extended with constraints. After the transformations described in Section 3, we linearize the patterns: if a pattern variable occurs multiple times in one pattern, we rewrite the pattern to use distinct variables, and we add an equality constraint. At this point, we can compile the pattern match into an efficient, bottom-up tree matcher in the style of BURG (Fraser, Henry, and Proebsting 1992).

We use a few tricks to improve the quality of the compiled matcher. A bottom-up tree matcher works by associating each subtree with a *state*. Such a matcher can be table-driven; the table is indexed by a tree constructor and by the states of the subtrees. One can compress tables by identifying sub-states that are equivalent, then merging table entries for such sub-states (Chase 1987; Proebsting 1992). Table-compression heuristics work best on matches in which common patterns have been factored out. To improve factoring in the matches we generate, we use the structure of operands in the λ-RTL description. For example, if an instruction takes as operand an addressing mode with eight alternatives, we do not expand the instruction into a list of eight patterns. Instead, we introduce a pattern-match nonterminal to stand for the addressing mode. We also keep code size down by introducing a single named function to stand for any fragment of code that is common to two or more actions.

6 Results

We have used the λ-RTL toolkit and our match compiler to generate a recognizer for the x86 target in our Quick C-- compiler, which is implemented in Objective Caml. We used a machine description that describes 630 instructions; it is 1,160 non-blank, non-comment lines of λ-RTL code. The generated recognizer replaces a hand-written recognizer which describes only the 233 instructions used in the compiler; it is 754 non-blank, non-comment lines of Objective Caml and BURG code.

The major effort of integrating the generated recognizer with the compiler involved correcting bugs in the hand-written code expander, which often produced incorrect RTLs. For example, the RTL that represented the x86's block copy instruction was

Recognizer	Compilation Time	Recognizer Fraction	Size
Hand-written	69.29 s	3.99%	189,504 B
Machine-generated, factored	64.23 s	0.69%	555,804 B
Machine-generated, expanded	65.73 s	0.56%	1,330,036 B

Table 1. Time and space measurements for hand-written and machine-generated recognizers.

incorrect, but because the hand-written recognizer accepted the incorrect RTL, the bug went undetected. Other bugs in the hand-written expander included missing effects on floating-point condition codes. These types of bugs may be less likely to appear in a machine description: the author of a machine description is free to focus on describing the machine accurately, instead of worrying about how to convert the compiler's intermediate representation to machine code.

To evaluate the quality of the generated recognizers, we compare three different recognizers: a hand-written recognizer, a machine-generated recognizer with the operands factored out, and a machine-generated recognizer with the operands expanded. Like the hand-written recognizer, the generated recognizers include only the instructions used in the compiler, and when we run the compiler on our test suite, all three recognizers match the same RTLs. For each recognizer, we measured the time spent compiling our test suite, the percentage of compilation spent in the recognizer as indicated by gprof, and the size of the stripped object file (Table 1).

Although all three recognizers are generated from variants of a BURG specification language, the machine-generated recognizers use a different match compiler, which generates faster recognizers. This match compiler, like BURG, precomputes state tables at compile-compile time; the hand-written recognizer uses a match compiler that, like iBurg (Fraser, Hanson, and Proebsting 1992), computes the state tables at run time. The use of different match compilers also helps to explain why the machine-generated recognizer with factored operands is almost three times the size of the hand-written recognizer. The size of each machine-generated recognizer includes the precomputed state tables, which are known to dominate the size of a bottom-up match compiler.

The size of the recognizer is further affected by the factoring of the BURG specification. A well-factored BURG specification can produce a smaller, more efficient recognizer, as demonstrated by the machine-generated recognizers: the factored recognizer is less than half the size of the unfactored recognizer. The hand-written recognizer benefits further because the original programmer carefully factored the BURG specification by hand, whereas the specification of the machine-generated recognizer is factored only over the operands.

7 Related Work

The technique of compiling RTLs using machine-independent optimizations with a machine-dependent code expander and a recognizer was developed by Davidson and Fraser (1984) and refined by Benitez and Davidson (1994). This technique is also used in `gcc`. It provides effective scalar, loop, and machine-level optimizations without requiring many machine-specific compiler passes.

Compiler writers have used "machine descriptions" for years, but the term is normally used loosely to mean "whatever information is needed to retarget my compiler." Recently, some other researchers have begun to use machine descriptions that have a declarative flavor. For example, Ceng et al. (2005) present a machine-description language that is similar in spirit, if not in syntax, to λ-RTL. They use a machine description to generate a BURG specification for instruction selection. After instruction selection, the code proceeds to a register allocator and a code emitter; the optimizer does not have the opportunity to exploit information about the semantics of machine instructions.

Tröger (2004) uses declarative machine descriptions to implement a dynamic binary translator. The translator uses two machine descriptions; for each effect of a guest-machine instruction, it finds host-machine instructions which execute that effect. Provided the semantics of parallel execution are preserved, the effects can be executed in sequence.

From the vast literature on bottom-up tree matching, we mention only papers that we have found directly relevant. Early work by Hoffmann and O'Donnell (1982) provides a useful overview of what are now common top-down and bottom-up tree-matching algorithms. Table-compression techniques for compile-compile-time state tables in a bottom-up tree matcher were developed by Chase (1987) and refined by Proebsting (1992). An alternative approach to bottom-up parsing is to perform shift-reduce parsing on the intermediate representation (Glanville and Graham 1978).

Pattern matching is built into many functional languages, which are typically implemented using top-down matching. Top-down matching works well on hand-written patterns with few alternatives and shallow nesting (Scott and Ramsey 2000), but a machine's instruction set has hundreds of deeply nested alternatives. For machine instructions, top-down matchers are prohibitively large, even when clever compression techniques are used (Eddy 2002).

8 Conclusion and future work

We have shown how to analyze a declarative, low-level machine description to recover the high-level abstractions used in a compiler. Leveraging these analyses, we generated a recognizer for the x86 in the Quick C-- compiler. The benefits of generating the recognizer are modest as long as the rest of the back end continues to be written by hand. Ultimately, we would like to generate *all* the machine-specific components of the back end, most notably, the code expander. Other plans include automatically checking the correctness of a λ-RTL machine description and incorporating elements of semantic matching to improve the flexibility of the recognizer.

Acknowledgements

Thanks to Paul Govereau, Glenn Holloway, and Kevin Redwine for helpful comments on early versions of this paper. This work has been supported by NSF grant CCR-0311482 and by an Alfred P. Sloan Research Fellowship.

Bibliography

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.

Jianjiang Ceng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. 2005 (March). C Compiler Retargeting Based on Instruction Semantics Models. In *DATE '05*, pages 1150–1155.

David R. Chase. 1987. An improvement to bottom-up tree pattern matching. In *POPL '87*, pages 168–177.

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

Jonathan Eddy. 2002 (April). A continuation-passing operator tree for pattern matching. Senior Thesis, Division of Engineering and Applied Sciences, Harvard University.

Lee D. Feigenbaum. 2001 (April). Automated translation: generating a code generator. Senior Thesis, Division of Engineering and Applied Sciences, Harvard University.

Mary F. Fernández and Norman Ramsey. 1997 (May). Automatic checking of instruction specifications. In *ICSE '97*, pages 326–336.

Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. 1992 (September). Engineering a simple, efficient code-generator generator. *ACM LOPLAS*, 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.

R. Steven Glanville and Susan L. Graham. 1978 (January). A new method for compiler code generation. In *POPL '78*, pages 231–240.

Christoph M. Hoffmann and Michael J. O'Donnell. 1982. Pattern matching in trees. *JACM*, 29(1).

Gerry Kane and Joe Heinrich. 1992. *MIPS RISC Architectures*. Prentice-Hall.

Christian Lindig and Norman Ramsey. 2004 (April). Declarative composition of stack frames. In *CC '04*, *LNCS* volume 2985, pages 298–312.

Todd A. Proebsting. 1992 (June). Simple and efficient BURS table generation. *PLDI '92,* in *SIGPLAN Notices*, 27(7):331–340.

Norman Ramsey and Jack W. Davidson. 1998 (June). Machine descriptions to build tools for embedded systems. In *LCTES '98*, *LNCS* volume 1474, pages 172–188. Springer Verlag.

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

Kevin Scott and Norman Ramsey. 2000 (May). When do match-compilation heuristics matter? Technical Report CS-2000-13, Department of Computer Science, University of Virginia.

Jens Tröger. 2004. *Specification-Driven Dynamic Binary Translation*. PhD thesis, Queensland University of Technology, Brisbane, Australia.