

**Automatically Generating the Back End of a Compiler  
Using Declarative Machine Descriptions**

A dissertation presented

by

João Dias

to

The School of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

December 2008

©2008 - João Dias

All rights reserved.

Thesis advisor  
**Norman Ramsey**

Author  
**João Dias**

## **Automatically Generating the Back End of a Compiler Using Declarative Machine Descriptions**

### **Abstract**

Although I have proven that the general problem is undecidable, I show how, for machines of practical interest, to generate the back end of a compiler. Unlike previous work on generating back ends, I generate the machine-dependent components of the back end using only information that is independent of the compiler's internal data structures and intermediate form. My techniques substantially reduce the burden of retargeting the compiler: although it is still necessary to master the target machine's instruction set, it is not necessary to master the data structures and algorithms in the compiler's back end. Instead, the machine-dependent knowledge is isolated in the declarative machine descriptions.

The largest machine-dependent component in a back end is the instruction selector. Previous work has shown that it is difficult to generate a high-quality instruction selector. But by adopting the compiler architecture developed by Davidson and Fraser (1984), I can generate a naïve instruction selector and rely upon a *machine-independent* optimizer to improve the machine instructions. Unlike previous work, my generated back ends produce code that is as good as the code produced by hand-written back ends.

My code generator translates a source program into *tiles*, where each tile implements a simple computation like addition. To implement the tiles, I compose machine instructions in sequence and use equational reasoning to identify sequences that implement tiles. Because it is undecidable whether a tile can be implemented, I use a heuristic to limit the set of sequences considered. Unlike standard heuristics, which may limit the length of a sequence, the number of sequences considered, or the complexity of the result computed by a sequence, my heuristic uses a new idea: to limit the amount of *reasoning* required to show that a sequence of instructions implements a tile. The limit, which is chosen empirically, enables my search to find instruction selectors for the *x86*, PowerPC, and ARM in a few minutes each.



# Contents

Title Page . . . . .	i
Abstract . . . . .	iii
Table of Contents . . . . .	v
Acknowledgments . . . . .	ix
<b>Preface</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Porting a compiler . . . . .	2
1.2 The big picture . . . . .	4
<b>2 Background</b>	<b>7</b>
2.1 Supporting many languages . . . . .	7
2.1.1 Targeting an existing virtual machine . . . . .	8
2.1.2 Targeting C . . . . .	9
2.1.3 Targeting C-- . . . . .	10
2.2 Compiler architectures and retargeting . . . . .	11
2.2.1 Dragon-book compilers . . . . .	12
2.2.2 Davidson/Fraser compilers . . . . .	13
2.3 Register-Transfer Lists (RTLs) . . . . .	17
2.3.1 State . . . . .	18
2.3.2 Instructions . . . . .	18
2.4 Declarative machine descriptions . . . . .	20
2.4.1 Details and abstraction in machine descriptions . . . . .	22
2.5 Summary . . . . .	24
<b>3 A compiler architecture for generating optimizing back ends</b>	<b>25</b>
3.1 The compiler's structure . . . . .	25
3.2 Automating compiler generation . . . . .	29
3.3 Generating the recognizer and expander . . . . .	31

<b>4</b>	<b>Recognizing RTLs using declarative machine descriptions</b>	<b>35</b>
4.1	Bridging the semantic gap . . . . .	36
4.1.1	Constants . . . . .	36
4.1.2	Locations . . . . .	38
4.2	Deciding equivalence efficiently . . . . .	43
4.2.1	Normal form . . . . .	44
4.3	Generating the recognizer . . . . .	45
4.3.1	Generating the match code . . . . .	46
4.3.2	Generating the matcher . . . . .	49
4.3.3	Evaluation . . . . .	50
<b>5</b>	<b>Instruction selection by machine-independent tiling of RTLs</b>	<b>53</b>
5.1	Tiling . . . . .	54
5.2	The model . . . . .	56
5.3	The tiles . . . . .	58
5.4	Specifying the tiler . . . . .	61
5.4.1	Data movement . . . . .	63
5.4.2	Computation . . . . .	66
5.4.3	Control flow . . . . .	69
5.4.4	Graph sequences and parallel assignments . . . . .	70
5.5	Possible extensions . . . . .	72
5.6	Properties of the tiler . . . . .	72
5.7	Obligations of the machine-dependent tileset . . . . .	73
5.7.1	Data movement . . . . .	74
5.7.2	Computation . . . . .	75
5.7.3	Control flow . . . . .	76
5.8	Tileset interface . . . . .	76
<b>6</b>	<b>Generating a tileset</b>	<b>79</b>
6.1	Techniques for finding tiles . . . . .	80
6.1.1	Algebraic laws and equivalence of expressions . . . . .	82
6.1.2	Compensating for extra assignments . . . . .	84
6.1.3	Data-movement graph . . . . .	85
6.2	The algorithm in action . . . . .	87
6.2.1	The shape of the algorithm . . . . .	90
6.2.2	Connecting the tileset and the tiler . . . . .	92
6.3	The algorithm . . . . .	92
6.3.1	The Hoare-logic judgment . . . . .	93
6.3.2	Model-theoretic account of the judgment . . . . .	94
6.3.3	Algorithm overview . . . . .	96
6.3.4	Establishment . . . . .	98

---

6.4	Discussion . . . . .	101
6.5	Pruning and termination . . . . .	102
6.6	Pragmatics of tileset generation . . . . .	106
<b>7</b>	<b>Search and termination in tileset generation</b>	<b>109</b>
7.1	Search Strategies . . . . .	111
7.2	Pruning Strategies . . . . .	111
<b>8</b>	<b>Costs and benefits of generating back ends</b>	<b>115</b>
8.1	Quality of generated code . . . . .	116
8.2	Ease of retargeting . . . . .	119
8.2.1	Quick C--: hand-written back ends . . . . .	120
8.2.2	Quick C--: generated back ends . . . . .	123
8.2.3	lcc . . . . .	126
8.2.4	vpo . . . . .	128
8.2.5	gcc . . . . .	130
8.3	Summary . . . . .	133
<b>9</b>	<b>Related Work</b>	<b>135</b>
9.1	Approaches to retargeting compilers . . . . .	135
9.2	Machine descriptions . . . . .	136
9.2.1	Domain-specific languages . . . . .	136
9.2.2	Reusable languages . . . . .	137
9.3	Generating code generators from machine descriptions . . . . .	138
<b>10</b>	<b>Conclusions</b>	<b>141</b>
10.1	Summary of results . . . . .	142
10.2	Contributions . . . . .	142
10.3	Future work . . . . .	143
<b>A</b>	<b>Automatically generating an instruction selector is undecidable</b>	<b>145</b>
A.1	The Instruction Equivalence Problem . . . . .	145
A.2	Proof of undecidability . . . . .	146
A.2.1	Defining a Turing machine . . . . .	146
A.2.2	Encoding Turing-machine states as instructions . . . . .	148
<b>B</b>	<b>RTL operators</b>	<b>157</b>
<b>C</b>	<b>Algebraic laws</b>	<b>161</b>
	<b>References</b>	<b>167</b>



## Acknowledgments

A dissertation may be the work of one person, but it cannot be completed alone. Many people have supported me over the years, both as colleagues and as friends.

Norman Ramsey introduced me to the world of research when I was an undergraduate, and he made it look like fun. Over several years as my advisor, Norman has provided patient but insistent guidance, encouraging me to produce the best possible work. I have enjoyed our work and our conversations immensely over the years.

I also thank my committee of David Brooks, Jack Davidson, and Greg Morrisett, who provided valuable feedback at various stages of my research. Mike Smith also offered helpful advice throughout my time as an undergraduate and graduate student.

During a pair of 6-month sabbaticals from my dissertation, Craig Chambers and Simon Peyton Jones provided opportunities to work in their worlds. Both are inspiring not only for their technical contributions but also for their boundless energy and enthusiasm.

I have been fortunate to work with an endless supply of interesting, fun people. I enjoyed many lunches and coffees with Matthew Fluet, Paul Govereau, Kim Hazelwood, Kelly Heffner, and Kevin Redwine. I also enjoyed many stimulating conversations with Lex Stein, who brought a unique perspective and vibrant sense of humor to any situation. Glenn Holloway may be the most helpful person on the planet, providing advice and practical assistance for any problem that might confront a graduate student. In addition to providing constructive criticism on my research, Glenn's help has saved me countless hours of work over the years.

I am also indebted to long-time friends both from college (especially Greg Kulesa, Adam Roberts, and Mike Tucker) and from graduate school (Jason Donald and Michael Hughes), with whom I enjoyed spending evenings away from work. And of course, I thank my family, who provided a supportive environment where education was a priority.

Most of all, I should thank my fiancée, Kathleen Guico, for her constant and patient support. Hard days of work were always easier when I could look forward to dinner with Kat. I owe her more than a few ice creams.



## Preface: notational conventions

In this preface, I define the notational and typographical conventions used in this dissertation. I also provide an index of notation.

### Notation for describing instructions

I describe an instruction using notation inspired by the publication language in the Algol 60 report (Backus et al. 1963), and by Bell and Newell's ISP (1971), with some modifications for describing machine instructions. An instruction computes the value of an expression  $e$  and assigns the resulting value to a location  $l$

$$l := e$$

But some instructions make parallel assignments to different locations; we compose two assignments in parallel using an infix vertical bar. For example, we can describe an instruction with two parallel assignments:

$$l_1 := e_1 \mid l_2 := e_2$$

Instructions may also be composed in sequence, using an infix semicolon. For example, we can describe two instructions composed in sequence:

$$l_1 := e_1 ; l_2 := e_2$$

This notation for instructions is similar to the notations developed by Dijkstra (1976) and Bell and Newell (1971), except that Bell and Newell use the semicolon for parallel composition.

Within an instruction, the definitions of locations and expressions differ depending on the exact language being described, but we maintain some notational conventions. A specific location is named using a typewriter font; for example, on the *x86*, the stack pointer is usually stored in the register `esp`. As in the examples described above, a metavariable  $l_i$  or  $e_i$  may be used to stand for a location or an expression, in which case the name of the metavariable is in italics, and a subscript may distinguish metavariables.

Operator name	Infix representation	Description
add	+	two's complement addition
sub	-	subtraction
mul	×	signed multiplication
quot	/	signed division, rounding toward 0
shl	≪	shift left
shra	≫ <sub>a</sub>	shift right, arithmetic
shrl	≫ <sub>l</sub>	shift right, logical
and	∧	bitwise and
or	∨	bitwise or
lt	<	less-than
gt	>	greater-than
conjoin	&&	logical and
disjoin		logical or

Table 1: Infix operators used in this dissertation

Not all metavariables are interchangeable: we may use one metavariable  $r_1$  to stand for an integer register and another metavariable  $f_1$  to stand for a floating-point register.

An operator application is usually described by first giving the operator in typewriter font, then giving a parenthesized list of subexpressions; for example, we add two expressions  $e_1$  and  $e_2$  using the notation `add( $e_1, e_2$ )`. Sometimes it is easier to read an application of a binary operator if it is written in infix notation; for example, the same addition expression can be written as  $e_1 + e_2$ . Table 1 lists the infix operators used in this dissertation.

Each expression has a type, which is either  $n$  bits or *bool*. Usually, we leave the types of the expressions implicit, but sometimes it is helpful to give an explicit type. If the type of a literal is notable, it may be given as a subscript; for example, a literal 7 represented using 12 bits is described as  $7_{12}$ . Similarly, the types of most operators are polymorphic in the widths of the operands, but because the types are usually apparent from context, we usually leave the types implicit. For example, the type of `add` is

$$\forall n : n \text{ bits} \times n \text{ bits} \rightarrow n \text{ bits}$$

When necessary, we specify the width of the operator using a subscript: `add32` adds 32-bit arguments.

```
1 max(x, y)  
2   if x > y then  
3     return x  
4   else  
5     return y
```

Figure 2: An example of a procedure in pseudocode

## Notation for describing algorithms

The remaining conventions determine the notation we use for pseudocode, as demonstrated by the example in Figure 2. The names of procedures and the names of data structures are always presented in italics. Keywords used in the pseudocode, such as **return**, are always presented in boldface.

The pseudocode is written in a vaguely procedural style: loops are described using iterators such as **forall**, and values are returned using an explicit **return** command. For reference in the text, I number each line of the example.

## Index of notation

While reading this dissertation, it may be useful to refer to the following index of the notations used throughout the dissertation. For each entry, I give the syntax, the page on which it is first introduced, and a brief description of what the notation represents. The index is divided into two parts: the first part lists metavariables, and the second part lists other syntactic forms.

Notation	Page	Description
$\emptyset$	63	empty set
$\oplus, \otimes$	18	operators
$b$	18	a Boolean constant
$e$	18	an expression
$\hat{e}$	58	a narrow expression
$g$	18	guarded assignment
$G$	57	a control-flow graph
$h$	58	a hardware register
$i$	18	an integer
$k$	18	a constant
$l$	18	a location
$ls$	94	a location set
$L$	58	a label
$\mathcal{L}$	61	locations written by extra assignments
$m$	58	a memory-like space
$n$	18	an integer
$r$	58	a register or temporary
$R$	18	an RTL
$s$	18	a storage space
$S$	63	equivalence (set)
$sl$	94	a singleton location
$\tau_e$	18	expression type
$\tau_l$	18	location type
$t$	58	a temporary (or pseudoregister)
$\hat{t}$	58	a temporary holding a narrow value
$x_e$	93	expression variable
$x_k$	93	compile-time constant variable
$x_l:ls$	93	location variable

Table 3: Metavariables used throughout this dissertation

Notation	Page	Description
$\triangleq$	63	equivalence (infix)
$\oplus(e_1, \dots, e_n)$	18	operator application
$\epsilon$	57	empty graph
$[\dots]$	46	match cost
$\langle \dots \rangle$	46	match constraint
$\{ \dots \}$	46	match action
$e \rightarrow l := e'$	18	guarded assignment
$e \succ e'$	90	$e$ is a subexpression of $e'$
$E[e]$	90	expression contexts
$g_1 \mid \dots \mid g_n$	18	parallel assignments (RTL)
$G; G'$	57	graph sequence
$G \stackrel{\mathcal{L}}{\subseteq} G'$	61	tiler refinement judgment on graphs
$G \stackrel{\mathcal{L}}{\subseteq}_M G'$	63	tileset refinement judgment on graphs
<b>goto</b> $e$	57	unconditional branch
<b>if</b> $e$ <b>then goto</b> $L_T$ <b>else goto</b> $L_F$	57	conditional branch
$l@n:\tau_l$	18	a sliced location
$n$ bits	18	Bit-vector width
$n$ loc	18	Location width
$nop$	18	instruction with no effect
$s[e]:\tau_l$	18	a location at address $e$ in the storage space $s$
$sl@n:\tau_l$	94	slice of a singleton location
$S \parallel S'$	63	disjoint sets
$S \cap S'$	63	set intersection

Table 4: Syntactic forms used throughout this dissertation



# Chapter 1

## Introduction

Writing an optimizing compiler has always been a difficult job, requiring deep knowledge of the source language, of compiler-construction techniques, and of the target machine. Ideally, we could divide this work among three different people: a programming-language designer could focus on syntax and semantics; a compiler expert could focus on register allocation and loop-invariant code motion; and a machine expert could focus on instruction-set architectures and pipelines. Achieving such a *separation of concerns* requires a new kind of compiler infrastructure in which support for a new language or a new target machine can be added with minimal effort. And of course, the compiler must generate efficient code for any combination of language and target machine.

An important contribution of this dissertation is to show how to separate machine knowledge and compiler knowledge, making it easier to add a new machine. A machine expert writes a description of the target machine's instruction-set architecture, independent of the compiler. And a compiler expert writes even low-level code-improving transformations in a machine-independent fashion. A compiler-compiler reads the description of the target machine and automatically generates the machine-dependent components of the back end. Compared to existing approaches for retargeting compilers, the tradeoffs are good for the machine and compiler expert: porting and writing the compiler are easier (as shown in Section 8.2). The tradeoffs for the user remain unchanged: the generated back ends produce code that is as good as the code produced by hand-written back ends. But this approach also introduces a new cost: the compiler-compiler requires sophisticated analyses and implementations.

The compiler-compiler generates a back end from a *declarative* description of the machine's instruction set. A declarative machine description describes a property of the instruction set, such as assembly syntax or in-

struction semantics, independent of any particular compiler or tool. In this dissertation, I show that we can analyze a declarative machine description and automatically generate the machine-dependent components of a compiler's back end. Although the problem of generating an instruction selector is undecidable in principle, it is possible in practice, and as distinct from previous work, the code produced by our generated back ends is as good as the code produced by hand-written back ends (as shown in Section 8.1).

Furthermore, by separating the machine knowledge from the compiler knowledge, I significantly reduce the difficulty of porting the compiler (see Section 8.2). The standard practice in porting a compiler requires a single person to understand both the structure of the compiler and the machine architecture, then to write the machine-specific components for the new back end. With my approach, porting a compiler requires an understanding of only the instruction-set architecture and the platform-specific conventions (e.g. standard calling conventions and stack layout). From a description of the target machine, most of the components of the back end are generated automatically, and a few small components that encode the conventions are written by hand. Because most of the back end is generated automatically, the compiler can be ported with significantly less effort: a new back end requires less code and can be written with a more superficial understanding of the data structures, algorithms, and invariants of the compiler.

In the following section, I expand on the standard approach to porting a compiler and give an overview of the better approach developed in the rest of this dissertation.

## 1.1 Porting a compiler

The state of practice in building portable compilers is depressing. Common practice is to clone and modify the machine-specific components of another back end, especially the instruction selector. In many compilers, the instruction selector is specified by a domain-specific language such as BURG (Fraser, Henry, and Proebsting 1992), in which the compiler writer writes patterns that map the compiler's intermediate representation to instructions on the target machine (Fraser and Hanson 1995; Poletto, Engler, and Kaashoek 1996; Burke et al. 1999; Castro 2001). Although a BURG-style specification is sometimes called a "machine description," it is really just a mapping from the compiler's intermediate code to machine instructions: an inseparable mix of compiler knowledge and machine knowledge.

Mixing the compiler and machine knowledge has substantial consequences: The BURG specification cannot be used for any purpose besides

selecting instructions for one particular compiler. And the BURG description can only be written by somebody who is an expert on both the target machine and the compiler's intermediate representation.

An important contribution of this dissertation is to show how to separate the machine knowledge from the compiler knowledge, making it easier to port the compiler to a new machine. I do so by automatically generating the machine-dependent components of the compiler from a declarative machine description. I use the SLED machine-description language, which describes the binary and assembly encodings of a machine architecture (Ramsey and Fernández 1997), and the  $\lambda$ -RTL machine-description language, which describes the semantics of machine instructions (Ramsey and Davidson 1998). Because a declarative machine description is not tied to any particular application, it can be used to generate components of many applications, such as binary rewriters (Cifuentes, Lewis, and Ung 2002), linkers (Fernández 1995), and debuggers (Ramsey and Hanson 1992), as well as compilers. And because a declarative machine description explicitly describes instructions, rather than defining a mapping from compiler representations to machine instructions, the machine description can be incorrect only if the machine expert describes the wrong semantics for the machine. A declarative machine description may be checked independently for correctness or consistency, as has been demonstrated with SLED (Fernández and Ramsey 1997); in future work, we hope to check the correctness of  $\lambda$ -RTL descriptions also.

In this dissertation, I have extended the  $\lambda$ -RTL toolkit (Ramsey and Davidson 1998) to analyze the SLED and  $\lambda$ -RTL machine descriptions and automatically generate the machine-dependent components of the compiler's back end, which are the instruction recognizer (Chapter 4) and the instruction selector (Chapter 6). Although I prove that the problem of generating an instruction selector is undecidable (Appendix A), heuristic search works well in practice. The instruction selector works by covering the compiler's intermediate code with a machine-independent set of tiles (Chapter 5); the search algorithm looks for sequences of machine instructions to implement these tiles (Chapter 6). The novel guiding principle of the search is to consider only sequences that consist of valid instructions on the target machine. Using this algorithm, I have generated back ends for the *x86*, PowerPC, and ARM architectures. And unlike the assembly code produced by automatically generated back ends in previous work, the assembly code produced by my generated back ends is as good as the assembly code produced by hand-written back ends (Chapter 8).

To understand this dissertation, you must understand the material covered in Chapter 2:

- The difference between the structure of a compiler described in the dragon book (Aho, Sethi, and Ullman 1986) and the structure developed by Davidson and Fraser (1984)
- Register-transfer lists (RTLs)
- $\lambda$ -RTL and SLED declarative machine descriptions

You must also understand the structure of our compiler, in which the main idea is to gradually establish stronger invariants on the RTLs that represent the intermediate code (Chapter 3).

## 1.2 The big picture

This dissertation shows that it is possible to automatically generate compiler back ends that generate good code, but it does not show whether this approach is cost-effective. The costs of my approach are clear; the benefits are less clear. Obviously, the more machines for which people want back ends, the more valuable my approach is. The period from 1985 to 1995 saw unparalleled architectural diversity, with the introduction of the MIPS, ARM, SPARC, 80386, HP PA-RISC, PowerPC, and Alpha. Retargeting was a pressing concern. Fifteen years later, the *x86* has emerged as the dominant architecture on the desktop. A skeptical reader might argue that reusability and portability are problems of the past: with only one target machine, implementing the infrastructure to generate the back end of a compiler is not cost-effective. But this argument misses the big picture:

- Interesting compilers are eventually ported to multiple architectures, justifying a one-time investment in infrastructure for generating back ends.

Although the *x86* is usually the initial target for a new compiler, interesting compilers are eventually ported to multiple target machines. In Figure 1.1, I list a number of compilers that are popular among procedural, object-oriented, and functional programmers; each of these compilers has been retargeted to new architectures as the number and needs of the users have grown. And we can expect greater interest in retargeting compilers to the rapidly expanding domains of embedded computing (Dubé and Feeley 2005), video-game consoles, and graphics processing (Fritz, Lucas, and Wilhelm 2007).

- I hope declarative machine descriptions can make it possible to generate a whole suite of machine-dependent tools. If the infrastructure

Compiler	Target machines	Year released
gcc	ARM, SPARC, x86, PowerPC, AVR, x64, ...	1987
vpo	VAX, MIPS, SPARC, x86, Motorola 88100, ...	1988
lcc	MIPS, SPARC, x86, Alpha	1988
SML/NJ	MIPS, SPARC, x86, HPPA, Alpha	1989
GHC	SPARC, x86, PowerPC, x64	1989
Ocaml	SPARC, x86, HPPA, PowerPC, Alpha, AMD64, ...	1996
Hotspot	ARM, SPARC, x86, PowerPC, x64	1999
MLton	SPARC, x86, HPPA, PowerPC, AMD64	1999
tinyc	ARM, x86	2006

Figure 1.1: Compilers that are popular among procedural (gcc, lcc, tinyc, vpo), object-oriented (Hotspot), and functional (GHC, MLton, Ocaml, SML/NJ) programmers have been ported to a variety of architectures. I list the compilers in the order in which development started, or the first release took place.

for generating these tools can be reused for multiple tool chains, the investment is worthwhile even with a single target machine.

In addition to the old standbys of compilers, assemblers, linkers, simulators, and debuggers, modern machine-level tools include dynamic optimizers (Bala, Duesterwald, and Banerjia 2000), just-in-time compilers (Ishizaki et al. 2003), binary rewriters (Tröger 2004), and tools for program-analysis such as Valgrind (Nethercote and Seward 2007). Similarly, new machine-dependent security tools are emerging, such as security-policy enforcement (Abadi et al. 2005; Erlingsson et al. 2006; McCamant and Morrisett 2006) and certified compilation (Leroy 2006). If we can use the same infrastructure to generate all these tools for use with multiple compilers, say GHC and MLton, then the investment of writing the infrastructure will pay for itself.

Of course, my work is just one part of the big picture. Previous work has shown how to automatically generate assemblers (Wick 1975), linkers (Fernández 1995), debuggers (Ramsey and Hanson 1992), program-analysis tools (Srivastava and Eustace 1994), and binary rewriters (Tröger 2004). In this dissertation, I automatically generate efficient back ends for an optimizing compiler.



# Chapter 2

## Background

A reusable compiler infrastructure must provide support not only for multiple target machines but also for multiple source languages. In this chapter, I explain the requisite background information to understand how our compiler, Quick C--, addresses both problems. In particular, I explain:

- Our compiler's source language, which is a portable assembly language that can be targeted by front ends for a variety of source languages
- The code-generation strategy developed by Davidson and Fraser (1984), which makes it easier to generate machine-dependent compiler components
- Register-transfer lists (RTLs), which is the intermediate representation used by our compiler
- The declarative machine descriptions I use to retarget the compiler

### 2.1 Supporting many languages

Suppose you have designed a new programming language, and you want to experiment with it. At first, you might build an interpreter to test some simple programs. But it is not good enough to have an implementation of your language; you need an implementation that is efficient enough to encourage others to use your language. Eventually, you need an optimizing compiler.

What is the best way to get an optimizing compiler? One approach is to build your own. You could write your own code generator, register allocator, and optimizations, not to mention the implementations of baroque

calling conventions. But that requires a lot of engineering effort, all of which has been done before in existing compilers. A better approach is to reuse an existing compiler by targeting an existing virtual machine, a low-level programming language like C, or C--.

### 2.1.1 Targeting an existing virtual machine

Compilers for high-level languages such as Haskell (Wakeling 1998) and Standard ML (Benton, Kennedy, and Russell 1998; Benton, Kennedy, and Russo 2004) have revealed the tradeoffs of targeting a virtual machine designed for another language such as Java or C#. The main advantage of targeting a virtual machine is that the virtual machine provides a number of services that don't need to be reimplemented: garbage collection, exception handling, and interoperability with other code that targets the virtual machine. The main disadvantage of targeting a virtual machine is that the virtual machine makes decisions with important cost tradeoffs that may not be a good match for the new source language. There are two types of decisions with important cost tradeoffs: decisions about the semantics of the virtual machine's bytecodes and implementation decisions for run-time services.

Two important design decisions of the bytecode language are the virtual machine's data model and the model of control flow. The virtual machine may require that the front end encode all data within a particular data model. For example, the Java Virtual Machine requires all non-primitive data to be encoded as objects of a particular format; the virtual machine then provides bytecodes to allocate, observe, and mutate the objects (Lindholm and Yellin 1999). Even if the source language does not use objects, it has to shoehorn its data representation into the object model (Benton, Kennedy, and Russell 1998). The virtual machine might also lack support for control-flow mechanisms used by the new source language. For example, the Java Virtual Machine lacks support for tail calls (Lindholm and Yellin 1999), which are essential for implementing functional languages. A compiler for a functional language targeting the Java Virtual Machine must jump through hoops to avoid overflowing the stack (Benton, Kennedy, and Russell 1998). Similarly, a new object-oriented language using novel criteria for method dispatch (Chambers 1992) must implement the method dispatch in terms of the virtual machine's control-flow primitives.

Even if the source language can be shoehorned into the virtual machine's bytecodes without too much effort, the source language may not match well with the cost tradeoffs made in the virtual machine's implementation of run-time services. For example, different garbage-collection policies may have

different performance characteristics depending on the allocation patterns of the programs (Stefanović, McKinley, and Moss 1999). If the virtual machine's garbage collector is tuned for an object-oriented language, it may not be a good match for the allocation patterns of a functional language. Similarly, the virtual machine's exception-handling policy may not match the new source language. For example, in a language like Modula-3 or Java, raising an exception is often viewed as a rare case that should not be optimized at the cost of slowing down the non-exceptional case (Chase 1992a,b). But in other languages such as Objective Caml, exceptions are a common method of control-flow, as demonstrated by their frequent use in the standard library (Leroy et al. 2004); therefore, a virtual machine that implements costly exceptions for a language like Java may result in bad performance for Objective Caml programs.

### 2.1.2 Targeting C

Another approach is to target a low-level programming language that does not impose a particular data model, garbage collector, or exception handler. The sensible choice is C (Bartlett 1989; Peyton Jones 1992; Tarditi, Lee, and Acharya 1992; Cejtin et al.). The tradeoff with targeting C is that the front end must still implement the high-level language features with little help from the C compiler.

By compiling to C, we can improve upon a number of disadvantages we identified with targeting a high-level virtual machine. One advantage of compiling to C is that the C compiler does not impose a high-level data model, which allows the front end to determine the layout of data objects. But the front end is still not free to place data anywhere; for example, it cannot place initialized data in the code section before the start of a procedure (Peyton Jones 1992). Another advantage of compiling to C is that the C compiler does not make policy decisions about the implementation of run-time services; instead, the front end is left to implement the run-time services explicitly.

The disadvantage of compiling to C is that the language does not provide the support necessary for building efficient high-level language features. For example, C compilers do not support proper tail calls, in part because C's support for variadic functions conflicts with tail-call optimization (Probst 2001). Furthermore, to implement run-time services such as precise garbage collection, exception handling, or stack inspection, the run-time system needs to be able to walk the stack to find garbage-collection roots or exception handlers. But only the C compiler knows the size of the stack frames and where the variables are stored. Because the run-time

system has no way to get this information from the C compiler, it cannot inspect the stack maintained by the C compiler. Instead, some front ends implement these features by constructing an explicit stack on the heap and using the C compiler to generate code only for basic blocks (Peyton Jones 1992). Others use so-called “conservative” techniques to inspect the stack (Boehm and Weiser 1988; Bartlett 1988).

### 2.1.3 Targeting C--

The best approach is to target C--, a portable assembly language that provides the advantages of C without most of the disadvantages (Ramsey and Peyton Jones 2000). C-- provides an assembly-level data model, control-flow primitives designed to support the control-flow constructs used in high-level languages, and a low-level run-time system that does not implement any run-time services but provides enough information for the front end to implement an efficient run-time system.

The data model in C-- is very simple: a program can access memory (the heap, the stack, uninitialized data, and initialized data) at the byte level and can use an unlimited number of virtual registers. The front end is free to choose the layout of data structures and map them onto virtual registers or memory.

C-- provides a number of control-flow primitives designed to support the implementation of control-flow constructs in a high-level language (Ramsey and Peyton Jones 2000). Along with the usual assembly-level primitives for local control-flow (if, computed goto), C-- implements several interprocedural control-flow primitives, including tail calls, *alternate returns*, and *stack cutting*. Alternate returns allow a procedure call to return to multiple different continuation points in the calling procedure. For example, a procedure can raise an exception by returning to an exception handler instead of the normal return continuation, which is how exception handling was implemented in CLU (Liskov and Snyder 1979). Stack cutting performs a constant-time control transfer to an existing frame on the stack. Stack cutting can be used to raise exceptions with little cost, as in Objective Caml.

Because a high level run-time system requires the ability to walk the stack and manipulate variables, C-- provides a run-time interface that provides functions to walk the stack and manipulate the local variables in each stack frame. The C-- compiler is responsible for layout of stack frames and deciding where to store variables. Therefore, only the C-- compiler knows the size of stack frames and where to look up the value of a variable. The front end for a high-level language can use the run-time interface to inspect the contents of the stack frame in the implementation of run-time services

such as exception handling and garbage collection (Peyton Jones, Ramsey, and Reig 1999; Ramsey and Peyton Jones 2000).

Using the control-flow primitives and the run-time system provided by C--, a front end for a high level language controls decisions about cost tradeoffs in the implementation of run-time services (Ramsey and Peyton Jones 2000). The C-- compiler is responsible for providing the rest: efficient code generation for a variety of target machines. In this dissertation, I explain how I automatically generate a back end for each new target machine through analysis of a declarative machine description.

Before we can discuss how a back end is generated, we must consider how the back end of the compiler is structured.

## 2.2 Compiler architectures and retargeting

Because it has been featured in many compiler texts (Aho, Sethi, and Ullman 1986; Appel 1998; Cooper and Torczon 2004), most people are familiar with a *dragon-book compiler*,<sup>1</sup> which consists of three main parts: a front end, which translates the source code to a language-independent, machine-independent intermediate representation; an optimizer, which improves the intermediate representation; and a code generator, which translates the intermediate representation to assembly code. Because the machine-dependent components are isolated in the code generator, the dragon-book compiler is supposed to be easily retargetable. Previous attempts to automatically generate the machine-specific components of a compiler have worked exclusively with dragon-book compilers.

But Davidson and Fraser (1980, 1984) developed a very different type of portable compiler built around a machine-independent optimizer that is nonetheless capable of manipulating machine instructions. It is easier to generate components for an efficient compiler if the compiler is constructed in the style of Davidson and Fraser.

In contrasting the dragon-book compiler architecture with the Davidson/Fraser compiler architecture, I focus on the process by which each compiler is retargeted to a new machine, and how this process inhibits or supports the automatic generation a compiler back end.

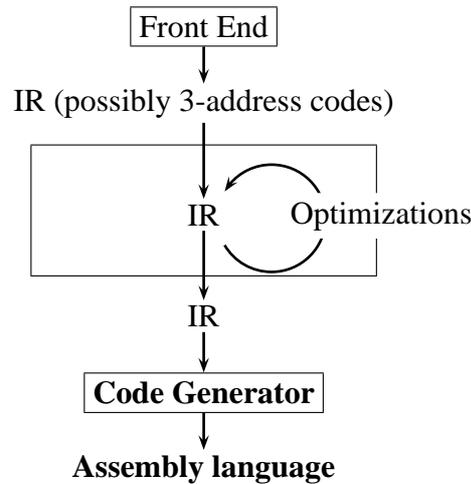


Figure 2.1: A dragon-book compiler: optimizations improve the language-independent, machine-independent intermediate representation, then the code generator produces assembly code. Machine-dependent components and representations are in **bold**.

### 2.2.1 Dragon-book compilers

The distinguishing feature of a dragon-book compiler (see Figure 2.1) is the language-independent, machine-independent intermediate representation. The intermediate representation allows the compiler to encapsulate language-dependent knowledge in the front end and machine-dependent knowledge in the code generator. And because the optimizer is independent of both source language and target machine, it can be reused for any combination of language and machine.

A dragon-book compiler is retargeted to a new machine by writing a new code generator<sup>2</sup>. The main components of the code generator are the instruction selector and the register allocator, but because the register allocator can be written in a largely machine-independent fashion (Smith, Ramsey, and Holloway 2004), I focus on the instruction selector.

The instruction selector is usually written using pattern matching over the intermediate representation. There are several ways to implement this pattern match, including bottom-up rewriting systems (BURS) (Aho, Gana-

<sup>1</sup>The phrase “dragon-book” refers to the cover of the classic compiler textbook by Aho, Sethi, and Ullman (1986).

<sup>2</sup>In practice, the compiler may reuse some parts of the code generator and require replacement of only the machine-dependent components.

pathi, and Tjiang 1989; Pelegrí-Llopart and Graham 1988; Emmelmann, Schröer, and Landwehr 1989), maximal munch, and ad-hoc code. Both maximal munch and BURS generate code using the same basic strategy: match the intermediate representation against a set of patterns, then execute a code fragment associated with the matched pattern. The code fragment generates the machine instructions for the intermediate code that was matched. A canonical member of the BURS family of code-generator generators is BURG (Fraser, Henry, and Proebsting 1992), which takes the patterns and code fragments as a specification, then generates an efficient bottom-up matcher that is guaranteed to produce locally optimal code. A related strategy is Graham-Glanville code generation (Glanville and Graham 1978), in which the intermediate representation is tokenized and parsed using an LR parser. Maximal munch (Cattell 1982) is a greedy, top-down algorithm that provides no guarantees but is easy to implement and works well in practice. In all of these strategies, one theme is constant: the instruction selector is written as a *mapping* from the compiler's intermediate representation to instructions on the target machine.

When a new target is added to the compiler, a brand new instruction selector must be written; the typical process is to clone an existing instruction selector for an existing target and modify it to use the instructions on the new target. Of course, if we want to modify the decisions made in the instruction selector for the old machine, we have to understand not only the compiler's intermediate representation and the new machine, but also the old machine. And because the instruction selector is an inseparable combination of compiler knowledge and machine knowledge, it cannot be tested independently for correctness.

We might attempt to automatically generate the instruction selector in a dragon-book compiler from machine descriptions, as others have done (Cattell 1982; Ceng et al. 2005). The problem with generating a code generator in this type of compiler is that the code produced by the code generator is the final code produced by the compiler. If the code generator produces inefficient code, the inefficiency is reflected directly in the compiler's output. The recent work by Ceng et al. (2005) resulted in code generators producing assembly code that ran about 5% slower than the assembly code produced by a hand-written code generator.

### 2.2.2 Davidson/Fraser compilers

Davidson and Fraser developed an architecture for writing portable, optimizing compilers without requiring the machine-dependent components to generate good code (Davidson and Fraser 1984). Central to Davidson's

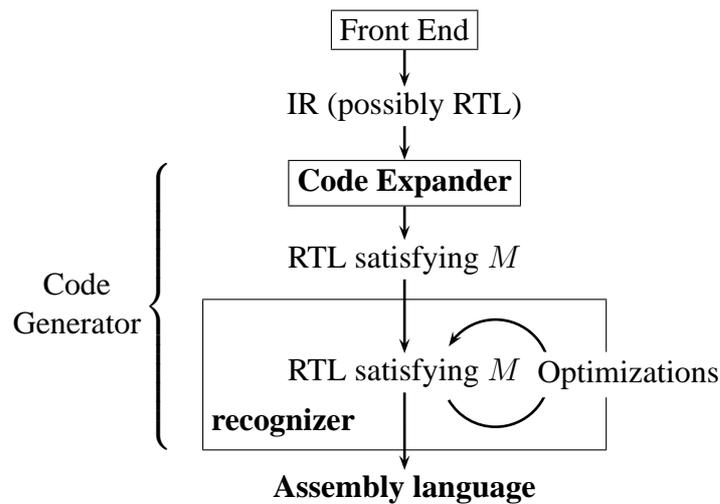


Figure 2.2: Davidson/Fraser compiler:  $M$  represents the machine invariant. Machine-dependent components and representations are in **bold**.

architecture is a machine-independent intermediate representation that can represent machine instructions: register-transfer lists (RTLs). Unlike many common intermediate representations such as 3-address codes, an RTL is not limited to evaluating finite expressions or mutating only a single location. For example, we can describe an instruction that evaluates a nested expression with a single RTL, such as the following RTL that describes a multiply-and-add instruction:

$$r_0 := r_1 \times r_2 + r_3$$

An RTL can also describe an arbitrary number of parallel assignments, as in the following swap instruction, which uses the infix operator  $|$  to separate parallel assignments:

$$r_0 := r_1 \mid r_1 := r_0$$

We describe the syntax and semantics of RTLs in Section 2.3 on page 17, but for now the important point is that RTLs can describe arbitrary machine instructions.

Using RTLs as the intermediate representation enables Davidson and Fraser’s compilation strategy: they choose machine instructions early and represent the instructions using RTLs, then use a machine-independent optimizer to improve the machine instructions. This strategy requires that the optimizer not only take machine instructions as input but also produce valid machine instructions as output. But how do you build a compiler that optimizes machine instructions without writing a new optimizer for each machine?

To this end, the compiler establishes the *machine invariant* on the intermediate representation: each RTL can be represented by a single instruction on the target machine. To check that an RTL satisfies the machine invariant, the compiler uses a *recognizer*, which is a machine-dependent component that tries to match an RTL to a single machine instruction. If a match is found, the recognizer can produce assembly code for the instruction.

In Davidson and Fraser's compiler architecture, shown in Figure 2.2, the compiler uses a phase called the *code expander* to transform the input code to RTLs satisfying the machine invariant. The expander translates a representation of the source program, such as

$$x = (y \times 4) + m[\text{stack}_z];$$

where  $m[\text{stack}_z]$  looks up a variable  $z$  on the stack, into a sequence of RTLs that can each be implemented by a single instruction on the target machine:

- A  $t_1 := 4;$
- B  $t_2 := y \times t_1;$
- C  $t_3 := \text{stack}_z;$
- D  $t_4 := m[t_3];$
- E  $x := t_2 + t_4$

The expander takes its name from the expansion of the input code into long sequences of instructions. Because the expander can rely on the optimizer to improve the code, it admits a simple implementation that produces a sequence of naïve instructions.

The optimizer improves the intermediate code, maintaining the machine invariant with help from the machine-dependent recognizer. Each optimization phase has a similar structure: a dataflow analysis identifies necessary facts for the optimization, then a transformation pass uses the results of the analysis to replace the old RTLs with more efficient RTLs. But before adding a new RTL to the control-flow graph, the transformation passes the RTL to the recognizer to verify that the RTL satisfies the machine invariant. If the new RTL does not satisfy the machine invariant, then the optimizer drops the new RTL and tries something else. To build intuition for how the optimizer works, we run through an example of peephole optimization on the code generated by the expander:

- A  $t_1 := 4;$
- B  $t_2 := y \times t_1;$
- C  $t_3 := \text{stack}_z;$
- D  $t_4 := m[t_3];$
- E  $x := t_2 + t_4$

Peephole optimization attempts to combine two instructions by replacing a location used in the second instruction by the value stored in the location by the first instruction.<sup>3</sup> For example, the value 4 computed in instruction A may be substituted for  $t_2$  in the instruction B:

$$t_2 := y \times 4$$

But the peephole optimizer is only allowed to keep the new instruction if the recognizer confirms that there is a machine instruction that multiplies the value in a register  $t_0$  and a literal. If we assume that the target machine includes instructions to multiply a register and a literal, fetch from a stack address, and add two registers, the peephole optimizer can produce the following sequence of instructions:

```
A   $t_1 := 4;$ 
B'  $t_2 := y \times 4;$ 
C   $t_3 := stack\_z;$ 
D'  $t_4 := m[stack\_z];$ 
E   $x := t_2 + t_4$ 
```

The peephole optimizer might attempt to substitute the value of  $t_4$  into the addition instruction E. But if the target machine does not have an instruction that adds a register to a value from memory, then the compiler is not permitted to represent the resulting RTL as a single instruction

$$t_1 := t_2 + m[stack\_z];$$

Consequently, the recognizer would reject the new RTL, and the peephole optimizer would move on to the next transformation.

After the forward-substitution pass, a subsequent dead-assignment elimination pass can remove RTLs that assign to dead locations, yielding the following code:

```
B'   $t_2 := y \times 4;$ 
D'   $t_4 := m[stack\_z];$ 
E    $x := t_2 + t_4$ 
```

Each of the optimizations follows the pattern of performing a semantics-preserving transformation, such as forward substitution, then using the recognizer to verify that the machine invariant is maintained. Davidson has shown how an optimizer with this structure supports not just peephole optimization but all the standard scalar and loop optimizations (Benitez and Davidson 1994).

---

<sup>3</sup>Depending on the implementation, the peephole optimizer may attempt to combine more than two instructions.

The advantage of the Davidson/Fraser compiler architecture is that the compiler can generate optimized code with simple machine-specific components: the expander generates naïve code and the recognizer matches machine instructions. Because the machine-specific components need not produce efficient code, it requires relatively little effort to retarget the compiler to a new machine. The disadvantage of the Davidson/Fraser compiler architecture is that it does not provide the separation of concerns we saw in the dragon-book compiler: although optimization passes can be written without concern for the source language or the machine, the language-dependent front end is required to establish the machine invariant, which is a machine-dependent property.

Because the expander requires language-dependent knowledge, we cannot hope to generate it from a machine description. But on the other hand, the Davidson/Fraser architecture relieves the machine-dependent components of the burden to produce efficient code, which means that generating even naïve implementations of the machine-dependent components will yield an optimizing compiler. In Chapter 3, we show how to adapt the Davidson/Fraser architecture to divide the expander into a machine-independent tiler and a machine-dependent tileset. The resulting structure takes advantage of the Davidson/Fraser optimization strategy while isolating the machine-dependent components, which allows us to automatically generate new back ends that produce efficient code.

## 2.3 Register-Transfer Lists (RTLs)

Like Davidson and Fraser, we use RTLs to describe instructions, both in our compiler's intermediate representation and in the  $\lambda$ -RTL machine descriptions. While the basic idea of RTLs is simple, a particular implementation may adapt the representation to the needs of the client. For example, an emulator does not need to distinguish between different types of storage: modeling a storage location should produce the same results regardless of whether the location is a register or a memory cell. But the distinction between registers and memory locations is important to a compiler because one of the compiler's most important responsibilities is to place frequently used values in registers. We describe in detail a representation of RTLs used in the machine descriptions and mention the relevant extensions used by the compiler.

A machine can be understood as state and a set of instructions that transform the state. First, we explain how the machine state is defined, then we describe how instructions are modeled using RTLs.

Loc types	$\tau_l ::= n \text{ loc}$
Exp types	$\tau_e ::= \text{bool} \mid n \text{ bits}$
Integers	$i, n$
Bool	$b ::= \text{true} \mid \text{false}$
Constants	$k ::= b \mid i$
Storage spaces	$s ::= 'a' \mid \dots \mid 'z'$
Locations	$l ::= s[e]:\tau_l \mid l@n:\tau_l$
Expressions	$e ::= k \mid l \mid \oplus(e_1, \dots, e_n)$
Guarded Assignment	$g ::= e \rightarrow l := e'$
RTL	$R ::= \text{nop} \mid g_1 \mid \dots \mid g_n$

Figure 2.3: A simple RTL grammar, including types for locations and expressions. The non-terminals  $i$  and  $n$  stand for integers and natural numbers, respectively. In most cases, the types are implicit and can be recovered using type inference.

### 2.3.1 State

The machine state is organized as a set of *storage spaces*. Each storage space is a named array of *cells*, indexed using array notation. For example, we might represent a set of registers using the storage space ‘ $r$ ’, and we can index the first register using the notation  $\$r[0]$ . A storage space is defined by four properties:

- *Cell width*: The number of bits in each cell in the storage space
- *Cell count*: The number of cells in the storage space
- *Aggregation*: Cells in a storage space may be aggregated in little-endian or big-endian order. For example, if a 32-bit number is stored in a memory space with 8-bit cells, the 32-bit number must be stored in four consecutive cells, but in what order? If the memory space is little-endian, then the least-significant byte is stored at the lowest address and the most-significant byte is stored at the highest address. If the memory space is big-endian, the bytes are stored in the opposite order. Besides memory spaces, aggregation is frequently used to represent double-wide values using register pairs. Some storage spaces do not permit aggregation, so we say their aggregation is the identity.

### 2.3.2 Instructions

An RTL describes how the machine state is mutated by an instruction. I present a simple RTL grammar in Figure 2.3, including the types of loca-

<i>Locations</i>	
$\$r[1]$	location 1 in the ‘r’ space
$\$r[rs1]$	location in $r$ space at address $rs1$ , where $rs1$ is a metavariable
$\$m[\$r[1] + i32]$	location in the $m$ space addressed by an offset $i32$ from $\$r[1]$
<i>Expressions</i>	
$\text{add}(\$r[rs1], \$r[rs2])$	addition of two registers
$\$r[rs1] + \$r[rs2]$	infix addition
<i>RTLs</i>	
$\$r[rd] := \$r[rs1]$	assignment from $\$r[rs1]$ to $\$r[rd]$
$\$r[rs2] < 0 \rightarrow \$r[rd] := \$r[rs1]$	guarded assignment predicated on $\$r[rs2]$ being negative
$\$r[rd] := \$r[rd] + \$r[rs1] \mid$	parallel assignment to register
$\$c[2] := \text{x86\_addflags}(\$r[rd] + \$r[rs1])$	and condition codes

Figure 2.4: Examples of the  $\lambda$ -RTL notation to describe RTLs. Contrast the  $\lambda$ -RTL code with the notation we use for presentation in Figure 2.3.  $\lambda$ -RTL provides type inference, which obviates the need to write types in most cases.

tions and expressions. A location must have the type  $n \text{ loc}$ , which means that the location is  $n$  bits wide. An expression may have either a Boolean type or a type  $n \text{ bits}$ , which means that the expression must evaluate to a bit vector of  $n \text{ bits}$ . Only an expression that produces a bit vector may be assigned to a location; an expression that produces a Boolean is used for describing either control flow or predicated instructions.

With the RTL types in mind, we can explain the locations and expressions used in RTLs. A location  $s[e]:\tau_l$  refers to a location in the storage space  $s$  indexed by the expression  $e$ . Because we aggregate cells implicitly, we have to make the type of the location explicit so that we know the width of the aggregate location, and hence, how many cells are aggregated. An RTL is well-formed only if the width of the aggregated location is a multiple of the width of the cells in the storage space. A location may also be a slice  $l@n:\tau_l$  of a location  $l$ . The slice is a smaller location contained within  $l$ , with least-significant bit  $n$  and width specified by the type  $\tau_l$ . A slice is well-formed only if the choice of least-significant bit and width defines a slice that is fully contained within  $l$ .

An expression  $e$  is either a fetch from a location, a constant, or the result of applying an operator to a sequence of subexpressions. An RTL may use two types of constants: booleans and bit vectors. We represent a bit vector simply as an integer  $i$ , with the understanding that  $i$  must fit in a fixed number of bits, which is specified by the expression's type.

Operator application deserves special mention because an operator may be polymorphic in the width of its operands and its result. For example, the addition operator  $+$  may be used to add two 16-bit arguments, or it may be used to add two 32-bit arguments. We say that the type of  $+$  is

$$\forall w : w \text{ bits} \times w \text{ bits} \rightarrow w \text{ bits}$$

which means that  $+$  takes two  $w$ -bit arguments and returns an  $w$ -bit result, for any width  $w$ . To simplify the presentation, we usually leave the types of the operators implicit.

A guarded assignment  $e \rightarrow l := e'$  describes the mutation of a location  $l$  with the value of an expression  $e'$ . But the assignment is executed only if the guard  $e$  evaluates to *true*. Guarded assignments are frequently used to describe conditional-branch instructions and predicated instructions, such as the conditional move instruction on the AMD64 architecture. If the guard is *true*, we often write the assignment without the guard.

An RTL is the parallel composition of zero or more guarded assignments, where  $|$  is the infix parallel-composition operator. A no-op is described by an RTL with no assignments: *nop*.

RTLs are used not only to describe instructions in a compiler but also to describe the semantics of a machine instruction in a  $\lambda$ -RTL machine description. In Figure 2.4 on page 19, I show examples of RTLs written in the syntax of  $\lambda$ -RTL.

## 2.4 Declarative machine descriptions

We describe a machine's storage spaces, instruction semantics, and instruction syntax using  $\lambda$ -RTL and SLED machine descriptions.

For each of the machine's storage space, the  $\lambda$ -RTL description specifies the number of cells in the storage space, the width of the cells in the storage space, the aggregation, and a descriptive string. For example, we describe the integer-register space and the memory space in the following  $\lambda$ -RTL declaration:

```
storage
  'r' is 8 cells of 32 bits called "registers"
  'm' is  cells of 8 bits called "memory"
      aggregate using RTL.AGGL
```

The aggregation RTL `.AGGL` for the *x86*'s memory space indicates little-endian order. If the aggregation is left unspecified, as in the *x86*'s integer-register space, then the cells in the space cannot be aggregated. If the number of cells is left unspecified, as in the *x86*'s memory space, then the number of cells is bounded by the width of the addressing expression.

$\lambda$ -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.

Using this model, a  $\lambda$ -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 assignments to condition codes (Section 2.4.1), 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]`.

The careful reader will note that the specification of the addition instruction does not mention any update to the program counter. In fact, the program counter requires careful modeling, as explained by Ramsey and Cifuentes (2003). The details are outside the scope of this dissertation, but the main result is that, from the compiler's perspective, we can treat the program counter like any other register. In instructions that increment the program counter to point to the next instruction, the update to the program counter is left implicit.

The *x86*'s call instruction is an example of an instruction with parallel assignments that also updates the program counter explicitly:

```
CALL.Jvod (reloc) is $c[0]           := reloc
                    | $m[$r[4] - 4] := $c[0]
                    | $r[4]         := $r[4] - 4
```

The program counter `$c[0]` gets the value of the label `reloc`, the program counter is saved just below the stack pointer `$r[4]`, and the stack pointer is pushed down to point to the saved program counter.

In similar fashion to  $\lambda$ -RTL, a SLED description associates each instruction with an assembly-language and binary representation. Given a SLED description, the  $\lambda$ -RTL toolkit can generate a function for each constructor that takes the operands as inputs and returns the assembly or binary encoding of the instruction (Ramsey and Fernández 1997). Because our work manipulates only the semantics of machine instructions, the details of SLED and the associated analyses are not necessary to understand our work.

### 2.4.1 Details and abstraction in machine descriptions

Real machines are often simple in the abstract but surprisingly complicated in detail. When people write machine descriptions, they do not want to specify the complete semantics of a piece of hardware: they want to write down just the parts they care about.

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. For each instruction adding two numbers  $x$  and  $y$  on the *x86*, the following parallel assignments are made to bits in the condition-code register:

```
SF := bit (x + y < 0) |
ZF := bit (x + y = 0) |
PF := bit (parity ((x + y)@bits[0..7]) = 0) |
OF := bit (add_overflows(x, y)) |
AF := carry(x@bits[0..3], y@bits[0..3], 0) |
CF := carry(x, y, 0)
```

But none of this detail is useful for a compiler writer. A common trick to simplify the description is to aggregate and abstract over multiple assignments. For example, I use a description of the *x86* that treats the condition-code register as an aggregate instead of as six individual bits. Each effect on the aggregate is described as the result of a machine-specific comparison operator. For example, to describe the addition instructions, I 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 assignments:

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

where the location EFLAGS in the second assignment is the register containing all the one-bit condition-code flags that are modified by the add instruction. The assignment to EFLAGS represents the same six parallel assignments to the condition codes, but the machine-specific operator helps us abstract away from the details.

Of course, the assignments to the condition codes are interesting only because they introduce assignments that are checked by other instructions. To use the results of an introduction construct like `x86_addflags`, I use an elimination construct; for example, to check whether the add instruction overflows, I introduce a new operator `x86_o` to check for overflow. The instruction to perform a conditional branch on overflow uses `x86_o` to check the condition codes in a guard:

```
Jv.0d (reloc) is x86_o(EFLAGS) --> PC := reloc
```

To conclude that an add instruction followed by this conditional-branch instruction will branch on overflow, I relate the two operators using an algebraic law:

$$\text{x86\_o}(\text{x86\_addflags}(x, y)) = \text{add\_overflows}(x, y)$$

This algebraic law exposes the most important property of the machine-specific operators: their composition checks for overflow.

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 the 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. For example, I introduce another machine-specific operator `x86_adcflags` to describe how an add-with-carry instruction modifies the condition codes, but if the carry-in value is 0, then the operator behaves the same way as the operator `x86_addflags`. The optimizer may miss opportunities to eliminate redundant add and add-with-carry instructions if it is not capable of concluding

$$\text{x86\_adcflags}(x, y, 0) = \text{x86\_addflags}(x, y)$$

## 2.5 Summary

In this chapter, I have explained the background information necessary for understanding the rest of this dissertation, focusing on the Davidson/Fraser compiler architecture, the language of RTLs, and the machine descriptions I use to generate compiler components. While reading the rest of this dissertation, it may be useful to refer back to this chapter, most notably the following parts:

- *Davidson/Fraser compiler architecture*: Figure 2.2 on page 14 shows the basic structure of a Davidson/Fraser compiler, which can be contrasted with Figure 2.1 showing the structure of a dragon-book compiler on page 12.
- *RTLs*: The RTL grammar in Figure 2.3 is on page 18.
- *$\lambda$ -RTL*: The correspondence between  $\lambda$ -RTL syntax and RTLs can be found in Figure 2.4 on page 19, and examples of  $\lambda$ -RTL code can be found on page 21.

## Chapter 3

# A compiler architecture for generating optimizing back ends

To retarget our compiler quickly and reliably, I generate the compiler back end from declarative machine descriptions. Previous efforts have generated dragon-book compilers (Cattell 1982; Ceng et al. 2005), resulting in back ends producing code that runs slower than a hand-written back end (notably a 5% slow down in recent work by Ceng et al. (2005)). In contrast, I use a Davidson/Fraser compiler, which allows me to generate machine-dependent components that produce naïve code, then rely on the optimizer to improve the code. The generated back ends produce code that is as good as the code produced by hand-written back ends. The price I pay for this architecture is that the instruction selector consists of not one component but two: the expander and the recognizer.

The rest of this chapter describes the structure of the Quick C-- compiler, with emphasis on the obligations not only for the expander and recognizer but also for the rest of the machine-dependent components. I then sketch my approach to generating back ends.

### 3.1 The compiler's structure

Our compiler is structured as a sequence of phases that gradually transform the source program to machine instructions. Figure 3.1 on page 26 shows the phases of the compiler, with source code flowing into the front end at the top and assembly code flowing out the code emitter at the bottom. We examine each phase in turn, with emphasis on how each phase changes the representation of the intermediate code.

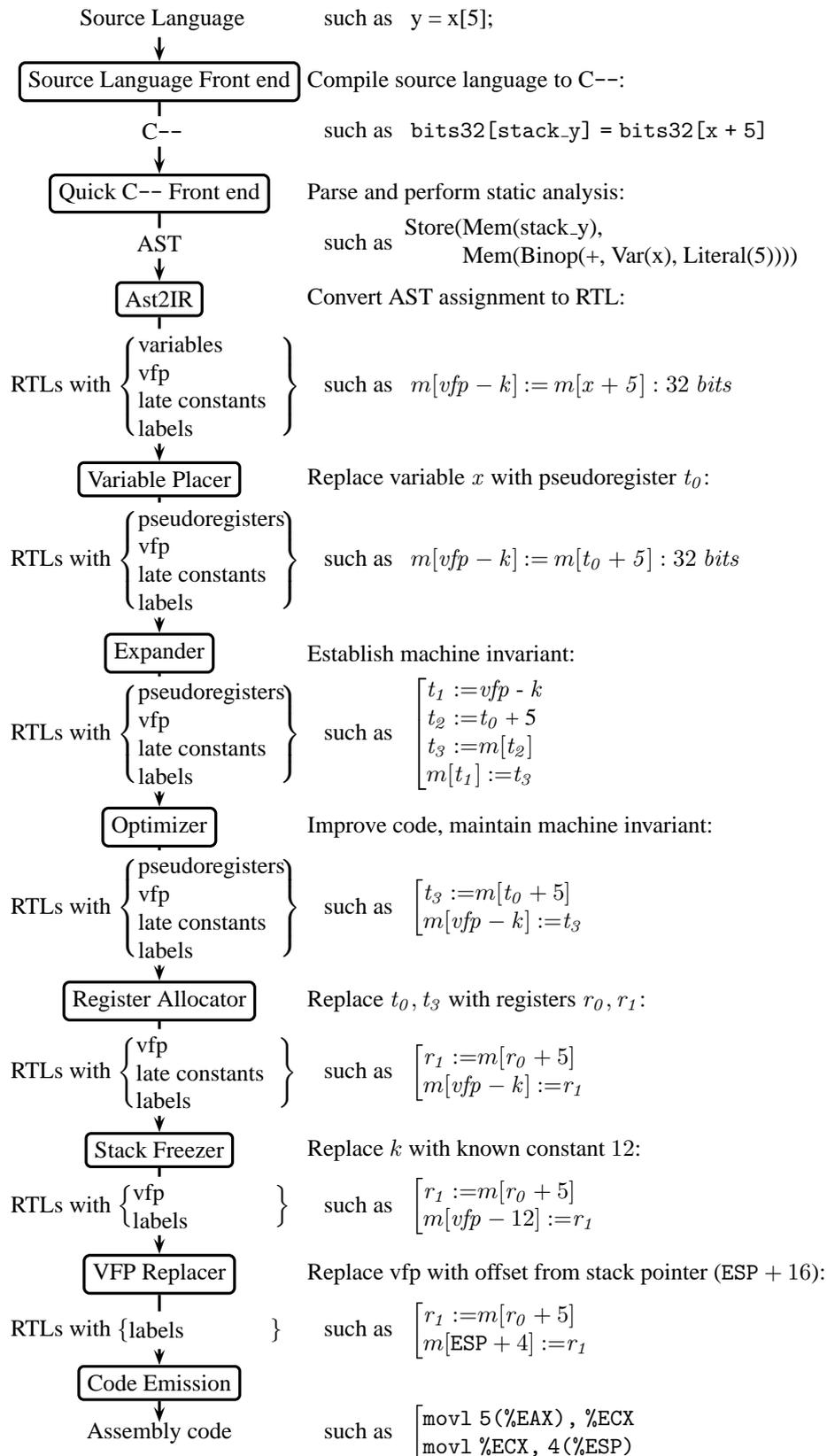


Figure 3.1: Translation by phases. Each phase is shown in a box; each arrow between phases describes the representation passed between those phases. The examples on the right show the evolution of a memory-to-memory move instruction.

Before our compiler is invoked, a front end such as `lcc` or `MLton` compiles a source program into the portable assembly language C-- . The front end of Quick C-- parses the C-- code, producing an abstract-syntax tree for each procedure. Each procedure is then passed in sequence to the back end of the compiler.

The first phase of the back end is the `ast2ir` phase, which performs a recursive walk over the abstract syntax tree, producing a control-flow graph with instructions represented as high-level RTLs (Ramsey and Dias 2005). The high-level RTLs may contain familiar representations of locations and constants, such as variables from the source language, stack slots, and labels. A stack slot is addressed using an expression  $vfp + k$ , where  $vfp$  is a *virtual frame pointer*<sup>1</sup> and  $k$  is a *late compile-time constant*. The  $vfp$  stands for the value of the stack pointer on entry to the procedure. Because the stack pointer may move during the procedure, especially when making tail calls using the standard C-- calling convention, it is inconvenient to address stack slots using the stack pointer. Instead, we address stack locations relative to the  $vfp$ . The  $vfp$  is *virtual* because each reference to it is eventually replaced by an offset from the stack pointer. Both the  $vfp$  and late compile-time constants are replaced after the stack layout is fixed. Before moving on, one additional transformation is applied to the RTLs by a component called the *widener* (Redwine and Ramsey 2004): if the program specifies computations that are narrower than the machine word, such as 16-bit addition, then the computations are replaced by equivalent computations at the width of the machine word.

The next phase of the back end is the *variable placer*, which puts each variable in the source program in a pseudoregister or a stack slot. The variable placer chooses where to place the variable depending on how the variable is used: if the variable is usually used in integer-arithmetic expressions, then the variable is placed in a new pseudoregister that can be used in integer-arithmetic expressions; similarly, variables used in floating-point expressions are placed in a new pseudoregister that can be used in floating-point expressions. To determine how a variable is used, the variable placer iterates over the control-flow graph and estimates the number of times a variable will be used for different purposes, such as integer arithmetic, floating-point arithmetic, or addressing a memory location. The variable placer replaces each variable with a pseudoregister that can be used in the expressions where the variable appears most frequently. For example, a variable that is used most frequently in floating-point computations is replaced with a floating-point pseudoregister. After the variable-placer phase, the RTLs

---

<sup>1</sup>The term *virtual frame pointer* may have appeared first in the MIPS assembler (Kane 1989), although in that context, the stack pointer did not move.

in the intermediate code may not contain variables, but they may contain pseudoregisters.

Of course, the sets of hardware registers, as well as the corresponding sets of pseudoregisters, are a property of the machine. Each target must tailor the variable placer to the available storage spaces. The specification for a target is usually just a few lines, consisting of simple policies such as widening an integer variable to the width of the machine's integer registers, then allocating the widened variable to a pseudoregister that stands for the integer registers. If the machine does not have registers to hold a particular kind of variable, the standard fallback is to place the variable on the stack.

The next phase is the expander, which establishes the machine invariant: provided that suitable substitutions are made for pseudoregisters, the virtual frame pointer, late compile-time constants, and labels, each output RTL represents a single instruction on the target machine. The expander is one of the two major machine-dependent components in our architecture, as it is responsible for selecting valid machine instructions. After the expander, the rest of the compiler is required to maintain the machine invariant. In most cases, the compiler maintains the invariant by rejecting any transformation that is not accepted by the other major machine-dependent component: the recognizer. But in some cases, the compiler needs to generate *new* instructions, such as loads and stores in the register allocator. For these cases, we provide a *machine record*, which provides functions for generating the simple instructions that may be generated in the rest of the compiler, such as loads and stores. Fortunately, the machine record can be defined as a collection of functions that invoke the expander, so defining a machine record for a new machine requires no extra effort.

After the expander establishes the machine invariant, the optimizer improves the code, maintaining the machine invariant with help from the recognizer (Section 2.2.2). The optimizer may use a series of passes to improve the code, possibly with repetition, but most optimizations have a similar structure: a dataflow analysis identifies necessary facts for the optimization, then a transformation uses the results of the analysis to replace the RTLs in the control-flow graph with more efficient RTLs. But before adding a new RTL to the control-flow graph, the transformation checks with the recognizer to verify that the new RTL satisfies the machine invariant. If not, the optimizer drops the new RTL and tries something else.

After optimization, the register allocator replaces the pseudoregisters with actual hardware registers. Of course, storage locations are a property of the machine, so the register allocator requires each target machine to provide a specification of the mapping from pseudoregisters to hardware registers. Our compiler provides both a linear-scan register allocator and the

iterated coalescing variation of graph-coloring register allocation (George and Appel 1996).

After register allocation, the compiler can finalize the layout of the stack in the *stack freezer*. Throughout the compiler, blocks of stack space are allocated for incoming function parameters, outgoing function parameters, private stack data, and spilled pseudoregisters. We specify the layout of the blocks using a declarative specification, which produces a system of equations relating the addresses of the blocks (Lindig and Ramsey 2004). The compiler solves the system of equations to determine the values of the compile-time constants that are used to address stack slots. The stack freezer then replaces the late compile-time constants with their computed values.

After the stack has been laid out, we can finally replace the virtual frame pointer with an offset from the stack pointer. The *vfp replacer* uses a simple dataflow analysis to compute the difference between the stack pointer and the *vfp* at each point in the procedure.<sup>2</sup> Then, each use of the virtual frame pointer is replaced with an offset from the stack pointer, where the offset is the difference between the stack pointer and the *vfp*. After the vfp replacer is finished, the remaining RTLs no longer contain any of the high-level representations used by the compiler (variables, pseudoregisters, compile-time constants, and the virtual frame pointer); with the exception of labels, which will be replaced by the linker, the RTLs now represent actual machine instructions.

The final stage of the compiler is the code emitter. The code emitter iterates over the control-flow graph a reverse postorder depth-first search to lay out the code, passing each RTL in the control-flow graph to the recognizer, which returns the assembly code for the corresponding machine instruction.

## 3.2 Automating compiler generation

To translate a program to machine instructions, the back end of the compiler uses a surprising number of machine-dependent components:

- *Instruction selection*: To choose machine instructions, the compiler uses a machine-dependent expander; to maintain the machine invariant, the compiler uses a machine-dependent recognizer.
- *Function calls*: To implement function calls, the compiler requires platform-specific calling conventions and stack-layout conventions.

---

<sup>2</sup>Recall that the stack pointer must move to support proper tail calls, so the stack pointer may move during a procedure.

- *Register allocation:* To place program variables in pseudoregisters, the compiler uses a machine-specific variable placer; to place pseudoregisters in hardware registers, the compiler uses machine-specific register specification to configure the register allocator.

My goal in generating a back end is to produce reliable compilers quickly, not to push automatic generation to its extreme. The most cost-effective way to build a compiler is neither to generate every machine-dependent component automatically nor to write every component by hand. I use a judicious mixture of three strategies:

- Write the component by hand in the implementation language of our compiler, Objective Caml (Leroy et al. 2004)
- Write the component by hand using a domain-specific language
- Generate the component automatically from declarative machine descriptions

I decide which strategy to use based on external constraints and engineering goals. The external constraints are conventions that cannot be inferred from a machine description. For example, we cannot possibly generate the C calling convention or its accompanying stack layout because neither are properties of the machine: they are established by human convention. The engineering goals are to reduce programming effort and to reduce opportunities for bugs: the code should be short and hard to get wrong. Of course, it can be difficult to argue that one type of code is inherently less error-prone than another. The criterion I use is to compare the types of machine and compiler knowledge required to write each type of code; I prefer the option that requires strictly less knowledge. As a consequence of these criteria, we make the following choices.

We write the specification of registers in Objective Caml. Although I could generate the list of available registers and their aliasing relationships as discussed by Smith, Ramsey, and Holloway (2004), we continue to write it by hand because the code is short and straightforward.

We write the calling conventions and stack-layout specification using domain-specific languages (Olinsky, Lindig, and Ramsey 2006; Lindig and Ramsey 2004) because it allows us to specify the conventions without the complication of writing compiler code. For example, using a domain-specific language allows us to specify the placement of calling-convention parameters without worrying about how the compiler generates the code to pass or receive the function parameters, which has caused numerous errors in other compilers (Bailey and Davidson 2003). Similarly, we specify the layout of

the stack without worrying about how the compiler keeps track of where data is stored on the stack and how the layout affects the code generated by the compiler. For the same reasons, we reuse the domain-specific language developed for calling conventions to specify how variables are placed in pseudoregisters.

Although the standard calling conventions cannot be generated from machine descriptions, I could probably generate native (non-standard) calling conventions automatically. I could then automate calling-convention experiments that others have performed by hand (Davidson and Whalley 1991) and generate a calling convention that performs well on a benchmark of choice. This possibility remains as future work.

The remaining machine-dependent components are the expander and recognizer, which collectively act as the instruction selector in a Davidson/Fraser compiler. Writing these components by hand is time-consuming and hard to get right; therefore, the only reasonable options are to use a domain-specific language like BURG or to generate the components automatically. But a domain-specific language requires us to understand both the compiler's intermediate representation and the machine's instruction set. Writing a declarative machine description, on the other hand, requires us to understand only the machine's instruction set. Because writing a machine description requires less intellectual effort while providing the benefits of reusability and reliability discussed in Section 1.1, I generate the machine-dependent expander and recognizer from declarative machine descriptions.

### 3.3 Generating the recognizer and expander

The main challenge in generating a recognizer is that the recognizer is used throughout the back end, which means that it must accept all the forms of RTLs that are used throughout the compiler. For example, before register allocation, an RTL may contain high-level representations of locations and constants such as pseudoregisters, the virtual frame pointer, and late compile-time constants. While these representations may seem very low-level when compared to a source language such as Java, these representations provide a very rich model of computation when compared to machine instructions, which manipulate nothing more abstract than a storage cell or a bit vector.

Besides labels, none of these representations are present in the machine description; they are useful for the compiler, but cannot be used in machine instructions. The back end of the compiler gradually maps these representations down to hardware locations and bit vectors. The recognizer must ac-

cept only those RTLs that will eventually be mapped down to real machine instructions. Therefore, to generate a recognizer, we analyze the machine descriptions to identify the set of RTLs that the compiler can map down to real machine instructions. In Chapter 4, we describe the analyses that identify the set of acceptable RTLs; in Section 4.3, we explain how we use the results of the analyses to generate compiler code for the recognizer.

The other major machine-dependent component is the expander, which is responsible for translating arbitrary RTLs to equivalent RTLs that represent machine instructions. To reduce the burden of writing (or generating) an expander, the expander is decomposed into machine-dependent and machine-independent parts using a novel type-based analysis of RTLs (Ramsey 2004). The result of the analysis is a set of *expansion tiles*, each of which is a small RTL schema with metavariables. Each tile performs one of three kinds of computation: data movement, application of an operator, or control flow. For example, the following tile represents two's-complement integer addition

$$r_1 := r_2 + r_3$$

where  $r_1$ ,  $r_2$ , and  $r_3$  are metavariables that stand for registers.

A machine-independent *tiler* takes a control-flow graph of well-typed RTLs and expands each RTL into a subgraph consisting of expansion tiles. Although the tiler and underlying analysis are not contributions of my dissertation, they are explained in Chapter 5, which also explains the set of expansion tiles that are required.

Each back end must provide a machine-dependent implementation of each expansion tile. An expansion tile is implemented as a function that accepts the tile's metavariables as arguments and returns a control-flow graph that implements the tile's RTL using only machine instructions. The automatic discovery of these implementations is the major contribution of this thesis; it is described in Chapter 6.

For convenience in implementing expansion tiles by hand, the interface used by the tiler actually has one function per *type* rather than one function per *tile*. The function takes an operator used in the tile as an additional argument, just as if it were a metavariable; the operator distinguishes the specific tile to be implemented. Because there are many fewer types than tiles, and because RTLs for similar machine instructions are often identical except for their operators, this design simplifies the interface considerably. For example, a single function that implements binary operators may have the type signature

**val** *binop* : *reg* × *operator* × *reg* × *reg* → *graph*

and implement multiple binary-operator tiles, as on the PowerPC:

Tile	Implementation on PowerPC
$\text{binop } r_{dst} + r_1 r_2$	$r_{dst} := r_1 + r_2$
$\text{binop } r_{dst} - r_1 r_2$	$r_{dst} := r_1 - r_2$
$\text{binop } r_{dst} \times r_1 r_2$	$r_{dst} := r_1 \times r_2$

When the tiler is used with the correct, machine-specific implementations of the expansion tiles, the result is a new control-flow graph whose observable input/output behavior is equivalent to the original graph, but which now satisfies the machine invariant.

The analyses used to generate the machine-dependent recognizer and to find implementations of the tileset are explained in the following chapters.



## Chapter 4

# Recognizing RTLs using declarative machine descriptions

The recognizer's job is to decide whether an RTL can be implemented by a single instruction on the target machine. Specifically, the recognizer must decide whether the RTL is *equivalent* to an instruction on the target machine. But deciding equivalence is not a trivial problem: two RTLs can be syntactically distinct but still have exactly the same effect on the state of the machine. Figure 4.1 shows several pairs of RTLs that are syntactically distinct but semantically equivalent.

Because the optimizer can only keep a transformed RTL if it is accepted by the recognizer, it would be ideal for the recognizer to accept any RTL that is semantically equivalent to a machine instruction. But because the recognizer is called with every RTL produced by the optimizer, it is in the compiler's inner loop, so it must be fast. And because checking for semantic equivalence can be time-consuming, the recognizer must be a compromise between speed and expressive power. The solution I explain in Section 4.2 is to generate a recognizer that performs syntactic matching of RTLs that have been reduced to a normal form. The resulting recognizer handles only the final example in Figure 4.1, but that example is particularly difficult to work around in the optimizer, and it is extremely common.

But there is another important complication to deciding whether an RTL can be implemented by a machine instruction: the compiler uses representations of constants and locations that are not present in a  $\lambda$ -RTL machine description. To generate a recognizer, we must bridge the *semantic gap* between  $\lambda$ -RTL's representation of instructions and the compiler's representation of instructions.

$r_1 := r_2$	$r_1 := r_2 \vee 0$
$r_1 := -r_2$	$r_1 := \text{com}(r_2) + 1$
$r_1 := 0$	$r_1 := \text{xor}(r_2, r_2)$
$r_1 := r_2 + r_3$	$r_1 := r_3 + r_2$
$r_1 := r_2 + 12_{32}$	$r_1 := r_2 + \text{sx}_{8 \rightarrow 32}(12_8)$

Figure 4.1: Each line shows a pair of syntactically distinct RTLs that are semantically equivalent.

## 4.1 Bridging the semantic gap

The recognizer has to recognize compiler RTLs, which may contain pseudoregisters and late compile-time constants, but we are given machine RTLs, which contain only hardware locations and literal constants. To bridge this semantic gap, we use a series of analyses to identify the set of compiler RTLs that the back end of the compiler will map down to instructions on the target machine. The basic idea is to compute the inverse image of the machine RTLs under the phases of the compiler’s back end. In practice, some analyses are nearly the inverse of compiler phases; others are not.

### 4.1.1 Constants

Our compiler uses two high-level representations of constants: labels and late compile-time constants.

#### Labels

The compiler uses a number of representations for constants: bit vectors, link-time constants, and late compile-time constants; the machine manipulates only bit vectors. We begin with the link-time constants, or labels, which are supported by the assembler and the linker. The compiler is free to take the simple approach of treating labels as absolute addresses, then rely on the assembler and linker to encode the label as an absolute or PC-relative address, as appropriate.

Our machine descriptions employ a similar separation of concerns for describing labels in machine instructions. The encoding of labels is managed by SLED (Ramsey and Fernández 1997), which can encode labels as either absolute addresses or PC-relative addresses. Because SLED can properly encode labels, the  $\lambda$ -RTL machine description is free to treat all labels

as absolute addresses, much like the compiler. For example, a relative jump on the *x86* can be encoded as an assignment to the instruction pointer:

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

The corresponding SLED description identifies the `addr` operand as a label. And because  $\lambda$ -RTL's description of labels coincides with the treatment of labels in the compiler, we do not have to make any special effort to identify how labels in the compiler's RTLs will map down to labels on the machine.

On some machines such as the PowerPC, the instructions can refer only to small constants (e.g. 16-bit constants); on these machines, the assembly language provides an addressing mode to extract the high or low bits of a label, and multiple instructions must be used to compose the entire 32-bit label. I encode these addressing modes faithfully in the machine description. As usual, SLED is responsible for properly encoding the labels, which means that the lower and upper bits of the labels are extracted as specified by the SLED description's definition of the addressing modes.

### Late compile-time constants

A late compile-time constant is a symbolic constant whose value is not determined until the stack layout has been frozen. When the stack is frozen, each late compile-time constant is replaced with a bit vector. Therefore, when a late compile-time constant appears in a compiler RTL, it is a proxy for a bit vector in a machine instruction.

But an instruction may not accommodate an arbitrary constant. In particular, each instruction limits the number of bits that are available for immediate operands. For example, while many *x86* instructions accept 32-bit immediate constants, the MIPS supports only 16-bit immediates, and the SPARC only 13-bit constants. We cannot know, a priori, whether a compile-time constant will resolve to a known constant that fits in the number of bits available in a machine instruction. So, how do we determine which compiler RTLs with late compile-time constants will map down to machine instructions?

One solution is pessimistic: if we cannot prove that a constant will be sufficiently small to fit in the instruction's immediate field, then we conclude that the constant cannot be used in the instruction. Instead, the compiler must find an equivalent sequence of instructions by breaking the constant into smaller bit vectors. For example, suppose the expander must produce code to compute an addressing expression  $sp + k$ , where  $k$  is a constant

that might not fit in the immediate field of the appropriate addition instruction. The expander can pessimistically assume that  $k$  will not fit, and issue a sequence of instructions that loads the high bits of  $k$  into the high bits of a register, loads the low bits of  $k$  into the register, and adds the value of  $sp$  to the register. After the stack layout is finalized, the optimizer may be able to remove extra instructions if it turns out that the constant  $k$  is small enough to fit in the immediate field of the addition instruction. But in the meantime, the compiler must deal with the extra instructions, which may have deleterious effects on phases such as the register allocator, due to increased register pressure.

A better solution is to assume optimistically that the late compile-time constants will fit in the immediate fields of the machine instructions. This solution leads to a simpler code expander and a smaller intermediate representation. In practice, most late compile-time constants are used to address stack slots, so the constants are usually small. When the optimistic assumption proves incorrect, the compiler must use a fix-up pass to use code that manipulates the offending constants in sufficiently small pieces. On some machines, the compiler may reserve a register for the fix-up pass. Sometimes, the assembler reserves a register and implements the fix-up pass, leaving the compiler free to manipulate large constants (Kane 1989).

### 4.1.2 Locations

Our compiler uses a number of high-level representations for locations: variables, pseudoregisters, and stack slots. A machine distinguishes between just two types of locations: registers and memory. Because variables are eliminated before the expander establishes the machine invariant, we do not need to consider how variables correspond to the locations that appear in machine instructions. But because our machine descriptions describe only the encoding and semantics of instructions, they need not make any distinction between storage locations; hence, our machine descriptions do not identify which locations are registers and which locations are memory. The compiler, on the other hand, needs to distinguish registers from memory because one of the compiler's most important tasks is to ensure that frequently used variables are stored in registers.

#### **Binding times**

Our understanding of how a compiler uses storage locations is based on a binding-time analysis developed by Feigenbaum (2001). The binding-time analysis determines when an expression's value becomes known:

```

ADDIdb (reg, i8)    is $r[reg] := $r[reg] + sx i8
                    | EFLAGS := x86_addflags($r[reg], sx i8)
MOVmr  (reg1, reg2) is $m[$r[reg1]] := $r[reg2]

```

Figure 4.2: Example instructions from the *x86*. The *MOVmr* instruction has been specialized to highlight the addressing expression in the destination; the actual specification of the instruction permits multiple types of addressing modes.

- *Specification time*: The value of the expression depends only on literal constants, so its value may be determined from the  $\lambda$ -RTL specification.
- *Instruction-creation time*: The value of the expression depends only on literal constants or the values of the instruction's operands, so it may be determined when the instruction is created. In the context of compilers, we usually refer to such an expression as a *compile-time expression*, since its value is known at compile time.
- *Run time*: The value of the expression depends on machine state, so it may not be determined until run time.

A simple analysis determines the binding time of each expression in the machine description by identifying whether the expression contains an operand or a fetch from storage.

Feigenbaum applies his binding-time analysis to the addressing expressions in a machine description to distinguish between different types of storage spaces:

- *Fixed space*: The value of each addressing expression is determined by the constructor used to build the instruction. For example, the *ADDIdb* instruction in Figure 4.2 modifies the condition-code register *EFLAGS* independent of the values of the operands. Therefore, the value of the addressing expression in the register *EFLAGS* is known at instruction-specification time. Because every addressing expression in the control-register space is known at specification time, the control-register space is a fixed space.
- *Register-like space*: The value of the addressing expression is determined by the constructor and the operands used to build the instruction. For example, the *ADDIdb* instruction in Figure 4.2 adds an immediate to an integer register, where the specific register is determined by the value of an operand. Therefore, the value of the addressing expression in the register is known at instruction-creation time (when

the operand is provided). Because every addressing expression in the integer-register space is known at instruction-creation time, the integer-register space on the *x86* is a register space.

- *Memory-like space*: The value of some addressing expressions may be determined by machine state. For example, the `MOVmr` instruction stores a value in a memory location that is addressed by the register `$r[reg1]`. The value of the addressing expression may not be known until run time. An obvious example of a memory-like space on the *x86* is the memory space; a somewhat surprising example on the *x86* is the floating-point stack, which the compiler addresses using the floating-point stack-top pointer.

Both fixed and register-like spaces are classified as registers; memory-like spaces are classified as memory. Using this classification, we can perform a simple analysis of the machine description to identify whether the compiler will treat locations in a particular storage space as registers or as memory.

The use of binding times to distinguish between registers and memory is not arbitrary; rather, it exploits a clever observation of why the distinction matters to a compiler. One of the essential tasks of a compiler is to allocate the fast but limited set of registers to frequently used values. To effectively allocate registers, the compiler must be able to place values in specific, individual registers. And to name an individual register, the compiler must be able to address the register at compile time. Therefore, the addressing expression of a register space must be a compile-time expression.

When the  $\lambda$ -RTL toolkit generates components for the compiler, we use the results of this analysis to generate the proper compiler representations for registers and memory locations.

### Pseudoregisters

A space of pseudoregisters represents an imaginary, infinite storage space that is distinct from any other storage space. Pseudoregister spaces are not one-to-one with spaces of hardware registers; instead, each space of pseudoregisters stands for a set of *interchangeable* hardware registers. For example, on the SPARC, the integer registers  $r_1$  to  $r_{31}$  are interchangeable in most instructions, while the register  $r_0$  is hardwired to the value 0. On the *x86*, the integer register space contains multiple sets of interchangeable registers: the 32-bit registers, the 16-bit registers, and the 8-bit registers all form distinct sets, even though they overlap.

The compiler uses pseudoregisters to stand for the real hardware registers that are used in machine instructions. The register allocator eventually

replaces the pseudoregisters with the hardware registers used in the machine instructions, inserting spill and reload instructions as necessary. By identifying the sets of interchangeable registers in the machine instructions, we can identify how the compiler may use pseudoregisters.

To identify candidates for pseudoregister spaces, we use a location-set analysis developed by Feigenbaum (2001). He classifies location sets into three categories:

- *Fixed location set*: A fixed location set contains only a single location. A fixed location set may appear in an instruction that refers to a specific location. For example, the multiplication instruction on the x86 refers specifically to the EAX register; no other location can replace EAX in this instruction.
- *Register-like location set*: A register-like location set contains a set of locations from a register-like storage space. An example of a register-like location set is the set of registers  $r_1$  to  $r_{31}$  on the SPARC.
- *Memory-like location set*: A memory-like location set contains a set of locations from a memory-like storage space. An example of a memory-like location set on the x86 is the memory space.

The analysis works by examining the addressing expression of each location in each machine instruction. If the value of the addressing expression is bound at specification time, then the location is the sole member of a fixed location set. If the value of the addressing expression is bound at instruction-creation time or run time, then the analysis assumes that the addressing expression may resolve to any bit vector of the appropriate width. For example, if the addressing expression is 32 bits wide, then the analysis assumes that the addressing may be any value in the range  $[0..2^{32})$ . The range of the addressing expression may be constrained by a guard on the RTL that contains the location. Many of the instructions on the SPARC require a register in the range  $r_1$  to  $r_{31}$ , so they use the guard  $\text{ne}(e, 0)$ , where  $e$  is the addressing expression.

In Figure 4.3 on page 42, we list the constraints derived from the SLED and  $\lambda$ -RTL descriptions, along with how they limit the range of the addressing expression. As more machine descriptions are written, they may use new constraints, requiring extensions to our simple constraint solver.

The resulting location set consists of those locations that fall within the possible range of the addressing expression. If the storage space is a register-like space, then the location set is a register-like location set; a memory-like space yields a memory-like location set.

Constraint	Range limitation
$e_w \neq i$	$[0..2^w) \setminus i$
$e_w < i$	$[0..2^w) \cap [0..i)$
$e_w \leq i$	$[0..2^w) \cap [0..i]$
$e_w > i$	$[0..2^w) \cap (i..2^w)$
$e_w \geq i$	$[0..2^w) \cap [i..2^w)$
<code>fits_unsigned(<math>e_w, i</math>)</code>	$[0..2^w) \cap [0..2^i)$

Figure 4.3: A constraint in an RTL’s guard may limit the range of a  $w$ -bit addressing expression  $e$ .

For each distinct register-like location set, the compiler may use a space of pseudoregisters. For each location in a machine RTL that belongs to a register-like location set, the compiler may use either a register from the location set or a pseudoregister in the corresponding pseudoregister space. But the  $\lambda$ -RTL toolkit must agree with the rest of the compiler on the names used to represent the pseudoregister spaces. We can either have the  $\lambda$ -RTL toolkit emit a file describing a mapping from register-like location sets to pseudoregister spaces, or  $\lambda$ -RTL can use an input file to specify the mapping. For use with our compiler, I provide an input file to  $\lambda$ -RTL that maps each location set with pseudoregisters to the name of the pseudoregister space.

When the  $\lambda$ -RTL toolkit generates components for the compiler, it uses the results of the location-set analysis to determine where an RTL may contain a pseudoregister. A generated recognizer must match either a hardware register in the location set or an appropriate pseudoregister; similarly, a generated expansion tile can produce RTLs containing either the appropriate hardware registers or pseudoregisters.

### Addressing stack slots

Throughout most of the compiler, a stack slot is addressed by a memory reference of the form  $m[vfp + k]$ , where  $vfp$  is the virtual frame pointer and  $k$  is a constant offset. As we saw in Chapter 3, the virtual frame pointer is eventually replaced with an expression of the form  $sp + k'$  where  $sp$  is the stack pointer and  $k'$  is another constant. Therefore, the compiler can use the virtual frame pointer in place of any expression in a machine instruction of the form  $sp + k'$ . But how do we know which register is the stack pointer?

The stack pointer is largely a matter of software convention;<sup>1</sup> therefore, the  $\lambda$ -RTL toolkit must be informed explicitly that a particular register will serve as the stack pointer. Given this information, we can conclude that the virtual frame pointer can be substituted for any expression  $l + k$ , where  $l$  is a location in the machine instruction that represents a location set containing the stack pointer.

When the  $\lambda$ -RTL toolkit generates components for the compiler, it permits any expression  $sp + k$  to represent the virtual frame pointer. A generated recognizer must match either the addition expression or a fetch from the virtual frame pointer; similarly, a generated expansion tile can produce RTLs containing either the addition expression or a fetch from the virtual frame pointer.

## 4.2 Deciding equivalence efficiently

The recognizer decides whether RTLs in the compiler's intermediate code are equivalent to machine instructions. Because the recognizer limits the transformations the optimizer can perform, it is important for the recognizer to accept as many RTLs as possible. Ideally, we would match two RTLs if they were semantically equivalent, that is, if executing the two RTLs effected the same change on the machine state. But computing semantic equivalence can be extremely expensive, and because the recognizer is used in the inner loop of the optimizer, it must be fast.

To ensure that the recognizer is as fast as possible, I generate recognizers that use matching to decide equivalence. There are two standard approaches to matching: simple non-linear pattern matches with guards<sup>2</sup> and sophisticated pattern matching that understands associative-commutative operators (Bachmair, Chen, and Ramakrishnan 1993). I implement straightforward syntactic pattern matching because it is fast and simple; the more complicated methods are too slow to run in the inner loop of the optimizer.

To implement syntactic pattern matching, I take each instruction in the  $\lambda$ -RTL description and transform it into a linear RTL pattern with guards. The recognizer accepts a compiler RTL as input and tries to match it to one of the RTL patterns representing machine instructions; if the RTL matches a pattern and satisfies any guard on the pattern, then the recognizer accepts the RTL.

---

<sup>1</sup>An instruction set frequently includes instructions that are useful for manipulating the stack pointer. If the instructions manipulate a fixed register, it is convenient to choose that register as the stack pointer.

<sup>2</sup>Functional languages like Haskell and Objective Caml provide built-in support for guarded pattern matching; BURS systems provide similar matching facilities.

But syntactic matching is very restrictive. A syntactic matcher cannot conclude that any of the RTLs in Figure 4.1 are equivalent. To improve the matcher, I follow a standard technique from the term-rewriting community (Bezem, Klop, and Roel de Vrijer 2003): I define a normal form for RTLs, then reduce each RTL to a normal form before trying to match it to the normalized machine RTLs. Of course, normalizing an RTL must also be fast, so the only transformation required to normalize an RTL is very simple: compile-time expressions must be reduced to constants.

### 4.2.1 Normal form

One of the most common ways in which the compiler generates semantically equivalent but syntactically different RTLs can be illustrated with the first assignment of the `ADDIdb` instruction on the `x86`:

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

A syntactic matcher will not match the `ADDIdb` instruction to an RTL such as  $r_0 := r_0 + 12_{32}$  because the 32-bit literal  $12_{32}$  is not syntactically equivalent to  $sx_{8 \rightarrow 32}(i8)$ . But we would like to match the RTL as a proxy for the semantically equivalent  $r_0 := r_0 + sx_{8 \rightarrow 32}(12)$ , which *is* a syntactic match. We can frame the problem in terms of binding times: *the recognizer should accept a constant in place of an expression that is bound at instruction-creation time.*

We solve this problem by reducing RTLs to our normal form, with cooperation from both the recognizer and the compiler. The recognizer promises to match a constant any time the machine instruction describes an expression bound at instruction-creation time, and the compiler promises to present only RTLs that have been normalized to replace compile-time expressions with constants.

#### Normalizing semantic descriptions of machine instructions

It is not safe for the recognizer to accept *any* constant in place of a compile-time expression. For example, the `ADDIdb` instruction should not match the literal constant  $32735_{32}$  because that value cannot be obtained by sign-extending an 8-bit immediate constant. To ensure that a constant can be computed by the expression in the original instruction, we use *constraints* to limit the value of the constant. For example, in the `ADDIdb` instruction, we rewrite the RTL to replace the compile-time expression `sx i8` with a variable `const` and a constraint:

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

The where constraint ensures that the value of the constant `const` can be obtained by sign-extending an 8-bit value, as in the original machine instruction. As discussed in Section 4.1.1 on page 38, we optimistically assume that the predicate is satisfied until we can simplify the expression to a literal value and evaluate the predicate precisely.

The transformation of the compile-time expression `sx i8 to const` has one more subtle consequence: the operand `i8` is no longer mentioned in the RTL. Because we need the operands to generate assembly code for the instruction, we must be able to reconstruct the value of `i8` from the replacement expression. In the `ADDidb` instruction, we can reconstruct `i8` by extracting the lowest 8 bits of `const`. The final result is an RTL with variables in place of compile-time constant expressions, constraints that maintain the original semantics of the instruction, and equations that reconstruct the values of operands that no longer appear in the RTL:

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

### Normalizing RTLs in the compiler

With the recognizer matching constants rather than compile-time expressions, the compiler must simplify all compile-time expressions to constants before passing an RTL to the recognizer. If all the leaves of a compile-time expression are literal constants, then the compiler can simplify the expression to a literal constant by evaluating the expression, which is sometimes known as constant folding. But a compile-time expression may contain late compile-time constants, in which case the expression cannot be evaluated to a literal constant. In that case, the compiler replaces the compile-time expression with a late compile-time constant as a proxy for the eventual value of the expression.

## 4.3 Generating the recognizer

The recognizer serves a dual purpose: it matches RTLs that can be represented by single machine instructions, and it returns the assembly code for the instructions. The recognizer satisfies both of these purposes by matching an input RTL to an instruction in the  $\lambda$ -RTL machine description, then using

the instruction encodings in the SLED specification to generate assembly code for the instruction. To generate the recognizer, first the  $\lambda$ -RTL toolkit uses the analyses in Section 4 to find the RTLs that should be matched by the recognizer. Then, the  $\lambda$ -RTL toolkit generates a BURG-style matcher for RTLs.

We use a BURG-style matcher both for its expressiveness and for practical reasons. BURG allows us to factor out the addressing modes of the instructions (Section 4.3.2 on page 49). In functional languages like Objective Caml, this factoring is not possible because you can only abstract over the pattern match on the operand if it is guaranteed to succeed. Furthermore, a previous effort to generate complicated pattern matches in Objective Caml resulted in a prohibitive blow-up in the size of the generated code (Eddy 2002).

### 4.3.1 Generating the match code

The  $\lambda$ -RTL toolkit uses the analyses in Section 4 to identify the set of RTLs that the recognizer should match. For each of these RTLs, we generate a pattern match in the domain-specific language we use to generate our BURG-style matcher. Each pattern match has five parts: a nonterminal, a tree pattern, a constraint, a cost, and an action:

```

nonterminal: tree pattern
<: constraint :> [ cost ]
{: action :}

```

A tree pattern is either a literal value, a nonterminal, or an operator applied to a sequence of subpatterns. If a pattern is a nonterminal, then it can match any pattern defined with the same nonterminal. A literal may be bound to a variable, in which case the variable is bound to the value of the literal. A nonterminal may also be bound to a variable (using the syntax `var : nonterm`), in which case the variable has different bindings in the guard and the action, as we explain below.<sup>3</sup>

The cost is a natural number associated with each pattern; if multiple patterns match, the one with the lowest cost is chosen. The total cost of matching a pattern is the sum of the pattern's cost and the cost of the patterns matched by nonterminals.

The constraint is a predicate written in arbitrary Objective Caml code. A pattern can only match if the constraint returns *true*. The constraint may refer to variables in the pattern, in which case the variable refers to the term being matched, *not* the action associated with the pattern. By binding

<sup>3</sup>I believe this dual-binding of variables is useful and unique among BURS systems.

the variable to the term, the constraint can reject a match based on the term being matched. For example, if two locations in an RTL should be the same, the constraint can verify that the locations being matched are equal.

The action is an arbitrary snippet of Objective Caml code that is evaluated if the pattern matches. An action may refer to a variable in the pattern, in which case the variable is bound to the result of evaluating the action associated with the pattern that matched the variable. The distinction between the variable binding in the constraint and in the action is essential to ensure that a constraint can check the input terms, while an action can use the results computed by the nonterminals in its pattern.

In a recognizer, the action constructs the assembly string for the matched instruction. A variable may be associated with a nonterminal that matches an operand, in which case the variable is bound to the assembly string for that operand.

The use of a match pattern is easier to understand with a concrete example. The following instruction is a 32-bit addition instruction on the *x86*:

```
ADDmrod (Eaddr, reg) is
  Eaddr := Eaddr + $r[reg]
  | $c[2] := x86_addflags(Eaddr, $r[reg])
```

Note that I have replaced the usual name for the condition-code register, `EFLAGS` with the actual location `$c[2]`. In Figure 4.4 on page 48, I show a tree representation of the add instruction, as well as the BURG patterns that match the instruction. The BURG pattern on lines 1-7 is a translation of the tree pattern into a text representation, with each node of the tree translated to a constructor in the BURG pattern. For example, the root of the RTL tree becomes the RTL2 constructor because the RTL makes two assignments. The add constructor takes not only the operands of the addition as arguments, but also the width of the addition (32 bits). The operands of the instruction, `reg` and `Eaddr`, are represented by the nonterminals `regloc` and `Eaddr`.<sup>4</sup> In the second set of BURG rules on lines 9-14, the `regloc` nonterminal matches any register in the ‘r’ space or the corresponding pseudoregister space ‘t’. The final set of rules on lines 16-21 defines the `Eaddr` nonterminal, which matches either a register matched by the `regloc` nonterminal or a memory location.

Each nonterminal is bound to a variable, which can be used in the constraint and the action. For example, the first tree pattern binds the variables `Eaddr1` and `Eaddr2` to the nonterminal `Eaddr`. In the constraint, the variable is bound to the part of the RTL that is matched by the tree pattern; for

<sup>4</sup>I have elided unnecessary details in the actual generated code: redundant widths and *true* RTL guards.

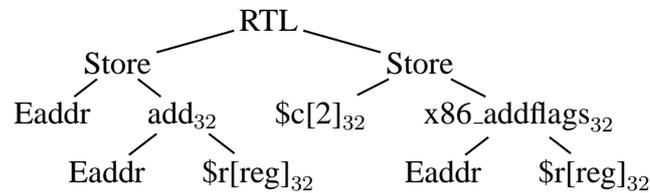
---

```

ADDmrod (Eaddr, reg) is Eaddr := Eaddr + $r[reg]
                    | $c[2] := x86_addflags(Eaddr, $r[reg])

```

---




---

```

1  inst: RTL2(STORE(Eaddr2:Eaddr, // match the ADDmrod instruction
2                      add(32, Eaddr1:Eaddr, reg0:regloc)),
3                      STORE(REG('c', intarg:int, 32),
4                      x86_addflags(Eaddr:Eaddr, reg:regloc)))
5  <: Base.to_bool (Loc.eq Eaddr Eaddr2 && Loc.eq Eaddr Eaddr1 &&
6                      Loc.eq reg reg0 && Int.eq intarg 2) :> [0]
7  : Instruction.ADDmrod Eaddr (Const.of_int reg 3) :
8
9  regloc: REG('r', reg:int, 32) // match registers in space 'r'
10 <: true :> [0]
11 {: reg :}
12 regloc: REG('t', reg:int, 32) // match pseudoregisters
13 <: true :> [0]
14 {: reg :}
15
16 Eaddr: r:regloc // match register
17 <: true :> [0]
18 {: Instruction.reg_atwidth_32 (Const.of_int r 3) :}
19 Eaddr: MEM('m', 32, Mem:Mem) // match memory location
20 <: true :> [0]
21 {: Instruction.e Mem :}

```

---

Figure 4.4: An example of a generated recognizer. The top panel show the definition of an add instruction in the  $\lambda$ -RTL description for the x86. The middle panel shows a tree representation of the same instruction, with the widths of operators and operands shown as subscripts. The bottom panel show the BURG patterns that match the instruction. The BURG patterns are a translation of the tree into a text representation, with separate patterns for the add instruction, the register operand, and the effective-address operand. The BURG patterns for the operands are reused for other instructions that use the same operands. I simplified the patterns for the add instruction to elide unimportant details (*true* RTL guards and redundant widths).

instance, the variable  $Eaddr_2$  is bound to the RTL location matched by the tree pattern. In the action, the variable is bound to the result of the action performed by the subpattern matched by the nonterminal.

As with most BURG-style matchers, our tree patterns must be *linear*: a variable may not appear more than once in the pattern. If an operand appears multiple times in an RTL, such as the operand  $Eaddr$  in the  $ADDmrod$  instruction, then the  $\lambda$ -RTL toolkit generates a fresh variable name for each occurrence of the operand in the tree pattern, along with an equality constraint. In our example, the constraint  $Loc.eq\ Eaddr\ Eaddr_2$  applies a function  $Loc.eq$  to compare two effective addresses for equality. Because the variables are bound to RTL locations in the constraint,  $Loc.eq$  is bound to a function that compares RTL locations.

The remaining fields of an instruction pattern are the cost and the action. The cost is a code snippet of Objective Caml, which computes the cost of the instruction. Because we compute the tables used for fast pattern matching at compile-compile time, the cost may not use any variables in the tree pattern. The action is a snippet of Objective Caml code that builds an assembly-language representation of the instruction using combinators in the `Instruction` module, which is generated by SLED (Ramsey and Fernández 1997). Each combinator in the `Instruction` module generates the assembly code for a constructor, as specified in the SLED description. The generated code is supplemented with a hand-written module `Const` that can emit proper assembly for constants. A standard `Const` module is provided with the  $\lambda$ -RTL toolkit, although it may require minor tweaking to specialize the assembly syntax to different targets.

### 4.3.2 Generating the matcher

We compile the match patterns into an efficient, bottom-up tree matcher in the style of BURG (Fraser, Henry, and Proebsting 1992). A bottom-up tree matcher works by associating each tree with a *state*; the state tree is determined by the tree constructor and the states of the subtrees. Such a matcher can be table-driven, with the table indexed by the constructor and the states of the subtrees. The tables can be computed at compile-compile time, which yields a fast matcher at compile time.

We can improve the quality of the compiled matcher by compressing the tables; we can compress tables by identifying substates that are equivalent, then merging table entries for such substates (Chase 1987; Proebsting 1992). Table-compression heuristics work best on matches in which common patterns have been factored out; otherwise, it is difficult to prove that two subtrees are interchangeable. To improve the factoring in the matches

we generate, we use the factoring provided by the  $\lambda$ -RTL description. For example, if an instruction takes an operand as 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; then each instruction that uses the addressing mode can use the nonterminal. 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.

Our matcher has one unusual combination of features: constraints in a matcher that computes tables at compile-compile time. If a matcher computes the cost of a pattern match at compile-time, then it can incorporate a constraint in the computation of the cost by returning an infinite cost if the constraint is not satisfied. But when the tables are computed at compile-compile time, constraints cannot be supported in the cost computation because the cost of a pattern match may not depend on the value being matched. Our matcher combines constraints with precomputed tables by making a minor modification to the algorithm. Normally, each table entry maps to a state. In our matcher, each table entry maps to a list of possible states in increasing order of cost. Each time the matcher performs a lookup in the table, it returns the first state for which the constraint is satisfied.

Because a bottom-up tree matcher's efficiency is a result of performing a linear walk of the input tree, a lookup table with long lists could degrade the performance of the matcher. In practice, we have not encountered this problem. The pattern matches produced by the  $\lambda$ -RTL toolkit do not have many layers of factored trees; therefore, because it is rare for a subtree to even have a constraint, constraints are usually checked when an entire instruction has been matched. And because very few instructions have identical tree patterns and differ only in their constraints, the number of potential states in a table entry rarely exceeds one or two. For example, on the *x86*, the state tables have only 12 entries with more than two potential states, and none of those entries has more than four potential states.

### 4.3.3 Evaluation

I have used the  $\lambda$ -RTL toolkit and our match compiler to generate a recognizer for the *x86*, PowerPC, and ARM targets in our Quick C-- compiler, which is implemented in Objective Caml. The machine description for the *x86* describes 630 instructions; it is 1,160 non-blank, non-comment lines of  $\lambda$ -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.

Recognizer	Compile Time	Recognizer Fraction	Size
Hand-written	69.29 s	3.99%	189,504 B
Generated, factored	64.23 s	0.69%	555,804 B
Generated, unfactored	65.73 s	0.56%	1,330,036 B

Table 4.5: Time and space measurements for hand-written and generated recognizers for the x86.

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 incorrect, but because the hand-written recognizer accepted the incorrect RTL, the bug went undetected. Other bugs in the hand-written expander included missing assignments 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 instructions.

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 unfactored. 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 4.5).

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. For example, on the *x86*, the hand-written back end defines special a tree pattern for the case of 2-assignment RTLs where one assignment sets the condition codes. When invoking the pattern matcher, the hand-written code identifies these RTLs and converts them to the specialized tree patterns before passing them to the matcher. Unfortunately, it is difficult to capture these patterns in a generated recognizer; although this is an interesting area for future work. Promising work on factoring match code by Eddy (2002) ran into practical difficulties in compiling the generated match code.

## Chapter 5

# Instruction selection by machine-independent tiling of RTLs

Besides the recognizer, the other major machine-dependent component in a Davidson/Fraser compiler is the expander. The expander transforms an instruction in the intermediate code into a sequence of machine instructions that implement the intermediate code.

To reduce the burden of writing (or generating) an expander, we factor the expander into machine-dependent and machine-independent parts. The interface between the two parts is a set of *expansion tiles*, each of which is a small RTL schema with metavariables. A machine-independent *tiler* takes a control-flow graph of well-typed RTLs and expands each RTL into a sub-graph consisting of expansion tiles. Each back end must provide a machine-dependent implementation of each expansion tile, in the form of a function that accepts the tile's metavariables as arguments and returns a control-flow graph that implements the tile's RTL using only machine instructions.

In this chapter, I explain how the tiler works and I specify the set of expansion tiles. Both the tiler and the set of expansion tiles were designed by Norman Ramsey, using a novel type-based analysis of RTLs (Ramsey 2004). Although neither the tiler nor the expansion tiles are my contributions, I describe them complete with a formal specification of the tiler, so that others may replicate our work. To understand how I automatically generate implementations of the tileset (Chapter 6), you need to understand only the basic tiling strategy (Section 5.1 on page 54) and the set of expansion tiles (Section 5.8 on page 76).

## 5.1 Tiling

Tiling is built around the idea of substituting trees for metavariables. Each metavariable has a type, which in the literature on tiling is usually called a *nonterminal*. A *tile* is a fragment of an intermediate-code tree or RTL, in which zero or more subtrees have been replaced by metavariables of given types. The tile itself is also given a type. A *tiling of type  $\tau$*  is defined inductively: it is a tile of type  $\tau$  in which each metavariable has been replaced by a tiling of the corresponding type.

Tiling is used in many code-generator generators such as BURG and Twig. Tiles are typically defined using a YACC-like notation in which the type of a tile is on the left-hand side of a colon, and the right-hand side is a tree fragment in which each metavariable is identified by its type. Here is an example from `lcc`'s `x86` code generator (Fraser and Hanson 2003), in which the type `reg` stands for an expression whose value can be computed into a register, and the type `mrc` stands for an expression whose value may be in a register, may be in memory, or may be a constant value:

```
reg: INDIRI4(VREGP) // register read
reg: ADDI4(reg,mrc) // add
reg: MULI4(reg,mrc) // multiply
```

`INDIRI4`, `VREGP`, `ADDI4`, and `MULI4` are all operators of `lcc`'s intermediate code. The first tile, which contains only operators and no metavariables, provides a base case for the inductive definition of a tiling.

A tile may consist of a single metavariable; such a tile is called a *chain rule*. For example, `lcc`'s `x86` code generator includes the following chain rules:

```
rc: reg
mrc: rc
```

The effect of the chain rules is that any tiling of type `reg` is also a tiling of type `rc`, and any tiling of type `rc` is also a tiling of type `mrc`.

A tiling is said to *cover* an IR tree or RTL if the tiling is identical to that IR tree or RTL. In this context the tiling is sometimes called a *covering*.<sup>1</sup> Given the rules above, we can conclude that the `lcc` intermediate-code tree for the sum of two registers,

```
ADDI4(INDIRI4(VREGP), INDIRI4(VREGP))
```

can be covered with tilings of type `reg`, `rc`, and `mrc`.

Instruction selection by tiling works as follows: we are given an IR tree and we find a covering; from the covering we then compute a sequence or

<sup>1</sup>The covering that is produced by a tree matcher like BURG is analogous to the parse tree produced by a string parser like YACC.

graph of machine instructions. There are two popular strategies for finding a covering:

- Associate a cost with each tile and use bottom-up rewriting to find a covering of minimum cost
- Use a greedy, top-down strategy to find a covering

To compute a graph of machine instructions in bottom-up rewrite style, each tile must be associated with an *action*, which is a fragment of code. Actions are executed bottom-up in a tiling, and the result of each tile's action is returned as the value of the corresponding metavariable in the parent's tile. For example, the tile for `lcc`'s `add` instruction specifies an action that constructs a sequence of two instructions: a move instruction and an add instruction:

```
reg: ADDI4(reg,mrc) "movl %0,%c\naddl %1,%c\n"
```

The results produced from the nonterminals `reg` and `mrc` are referred to by the names `%0` and `%1`. The name `%c` refers to a destination register for the addition, which is provided later by the code generator.

Recapitulating, instruction selection by tiling works as follows:

- Define a set of tiles sufficient to cover any well-formed IR tree
- Associate each tile with an appropriate action
- Select machine instructions by executing actions bottom up

### How our tiler is different

As tiling is used classically, one tile corresponds to one instruction or one addressing mode. Each machine requires a different set of tiles, with the actions written by hand. Although a set of tile-action pairs is often called a “machine description,” it is actually a recipe for an instruction selector. It is difficult to extract any other useful information from the tile-action pairs. To see if the set of tiles is sufficient and the actions are correct, compiler writers reason informally and run tests. Because each machine uses a different set of tiles, the reasoning has to be repeated for each machine.

As we use tiling, one tile corresponds to one construct in the IR. More specifically, a tile corresponds to the application of a single RTL operator, or to the movement of a single bit vector, or to a single control-flow operation. There are only two sets of tiles: register machines share one set of tiles, and stack machines share another set of tiles. Actions may be written by

hand but can sometimes be shared among machines. For example, machines that have 3-address ALU instructions with no effect on condition codes can share actions for the “binop” tiles: the actions for the addition tiles on both the SPARC and the PowerPC can produce the following RTL representing the add instruction

$$r_1 := r_2 + r_3$$

In this dissertation, I show how actions can be discovered automatically by analysis of a declarative machine description (Chapter 6). The same machine description can be used for other purposes, such as generating an emulator. As in the classical case, to see if the set of tiles is sufficient and the actions are correct, we reason informally and run tests. But because we have only two sets of tiles, we get to reuse the same reasoning for new machines.

## 5.2 The model

To identify the set of tiles, we analyzed the grammar of RTLs, paying particular attention to the RTL operators and their types. For example, almost all the integer and logical operators take two operands of width  $w$  and produce a result of width  $w$ . Each of these operators is implemented by a tile of the form  $r_1 := r_2 \oplus r_3$ . The integer-multiply operator, by contrast, takes operands of width  $w$  and produces a result of width  $2w$ , so its tile has a different form: it takes its operands in registers and puts its double-width result into a *pair* of registers  $r_{hi}, r_{lo} := r_1 \otimes r_2$ . A unary operator like bitwise complement has yet another form:  $r_1 := \oplus(r_2)$ .

Given our library of over 80 RTL operators, plus assignment and control transfer, there are dozens of tiles. But the number of tiles is less important than 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; shapes are important because if there is already a tile of the appropriate shape, it is trivial to add a new tile or a new RTL operator to our system.

The design of the tiles is driven by the classification of operators:

- A *standard-value operator* takes some bit vectors of reasonable width and returns a result of that width.
- A *weird-value operator* takes some bit vectors of reasonable width, but also may take an argument or return a result that is unusually narrow.

An important special case of weird-value operator is a floating-point operator, which takes ordinary floating-point operand(s) plus a rounding mode. The other weird-value operators are extended multiplies which double with width of their operands (`mulx`) and multiprecision operators such as computing the carry bit from an addition (`carry`), computing the borrow bit from a subtraction (`borrow`), adding with a carry bit (`addc`), and subtracting with a borrow bit (`subb`).

- A *size-changing operator* widens or narrows a bit vector. The integer size-changing operators are sign-extension (`sx`), zero-extension (`zx`), and extracting the low bits (`lobits`). The floating-point size-changing operators are conversion from integer to float (`i2f`), conversion from float to integer (`f2i`), and conversion between floats of different widths (`f2f`).
- A *comparison operator* takes one or more bit vectors and returns a Boolean. This category includes not only the usual integer and floating-point comparisons but also some machine-dependent tests of the condition codes. For example, the machine-specific operator `x86_o` operator (Section 2.4.1 on page 23) checks for overflow of an addition operation.
- A *Boolean operator* takes one or more Booleans and returns a Boolean. The Boolean operators are `conjoin`, `disjoin`, and `not`, as distinct from logical `and`, logical `or`, and bitwise complement (`com`). The tiler compiles them to control flow.

This classification describes the properties of the operators themselves. As such, it is orthogonal to the distinction between a register machine and a stack machine, which describes how operators are implemented on particular hardware.

In addition to an analysis of the operators, we also considered the form of control flow in our compiler, but because there are relatively few forms of control flow, the results are straightforward. Our treatment of the tiler in this chapter deals only with sequences, branch instructions, and conditional-branch instructions. For simplicity, we omit `call` and `return` instructions, which are handled in a similar fashion. Consequently, we consider only

$t$	a temporary
$\hat{t}$	a temporary holding a sign-extended or zero-extended value
$h$	a hardware register
$r$	a hardware register or a temporary
$\hat{r}$	a register holding a sign-extended or zero-extended value
$m$	a memory-like space
$l$	a location
$k$	a compile-time constant
$L$	a label or link-time constant
$\oplus, \otimes$	RTL operators
<b>ext</b>	a widening operator (gx, sx, or zx)
$e$	an expression whose width is the machine word
$\hat{e}$	an expression whose width is less than the machine width (e.g. a carry bit)

Figure 5.1: Notational conventions used to describe tiles.

four forms of graphs:

Graphs	$G ::= \epsilon$	empty graph
	$L :$	label
	$R$	RTL
	$G; G'$	sequencing
	<b>goto</b> $e$	unconditional branch
	<b>if</b> $e$ <b>then goto</b> $L_T$ <b>else goto</b> $L_F$	conditional branch

In the conditional-branch statement, the program jumps to the label  $L_T$  if  $e$  is true and  $L_F$  otherwise.

### 5.3 The tiles

We summarize the three different kinds of tiles we have identified, with tables showing the shape of the tile, the name of the tileset function implementing the shape, and a short description of the shape. The tile definitions use the notational conventions in Figure 5.1.

- A *data-movement tile* moves a constant into a location or moves a value between two locations. Figure 5.2 lists the shapes of data-movement tiles.
- A *computational tile* applies an RTL operator to constant or register operands and places a result in one or more registers. Recall that a

$r_1 := r_2$	move	register-to-register move
$r_1 := m[r_2]$	load	load from memory
$m[r_1] := r_2$	store	store to memory
$m[r_1] := m[r_2]:w \text{ bits}$	block_copy	memory-to-memory move
$r := k$	li	load immediate
$r := h$	hwget	fetch from hardware register
$h := r$	hwset	store to hardware register
$r_1 := \text{sx}(m[r_2]:w \text{ bits})$	sxload	sign-extending load
$r_1 := \text{zx}(m[r_2]:w \text{ bits})$	zxload	zero-extending load
$m[r_1]:w \text{ loc} := \text{lobits}(r_2)$	lostore	store least-significant bits
$r_1:w \text{ loc} := r_2@lo:w \text{ bits}$	extract	copy slice to a smaller register

Figure 5.2: Data-movement tiles store constants in locations and move data between locations.

$r_1 := \oplus(r_2)$	unop	a unary ALU operation
$r_1 := \oplus(r_2, r_3)$	binop	a binary ALU operation
$r_1 := \oplus(r_2, \hat{r}_3)$	unrm	a unary floating-point operation or conversion
$r_1 := \oplus(r_2, r_3, \hat{r}_4)$	binrm	a binary floating-point operation
$r_1 := \oplus(r_2, r_3, \hat{r}_4)$	wrdop	a weird-value operation, like add with carry
$\hat{r}_1 := \text{ext}(\oplus(r_2, r_3, \hat{r}_4))$	wrdrop	a weird-value/result operation, like carry
$r_{hi}, r_{lo} := \otimes(r_1, r_2)$	dblop	an extended multiply operation

Figure 5.3: A computational tile applies an RTL operator to constant or register operands.

<b>goto</b> $L$ b	branch
<b>goto</b> $r$ br	branch register
<b>if</b> $\oplus(r_1, r_2)$ <b>then goto</b> $L_T$ <b>else goto</b> $L_F$ ; $L_F$ :	bc branch conditional

Figure 5.4: A control-transfer changes the flow of program execution.

$m'[STP] := m[r] \mid STP := STP - 1$	push push onto stack from memory space $m$
$m'[STP] := \oplus(m[r]) \mid STP := STP - 1$	push_cvt convert to float or int and push
$m[r] := m'[STP] \mid STP := STP + 1$	store_pop pop from stack to memory space $m$
$m[r] := \oplus(m'[STP]) \mid STP := STP + 1$	store_pop_cvt convert floating type and pop from stack to memory space $m$
$m'[STP] := k \mid STP := STP - 1$	pushk push constant onto stack
$m'[STP] := \oplus(k) \mid STP := STP - 1$	pushk_cvt convert const to float or int and push onto stack
$m'[STP] := \oplus(m'[STP], \hat{r})$	stack_op_rm unary operator application
$m'[STP] := \oplus(m'[STP], m'[STP + 1], \hat{r})$ $\mid STP := STP + 1$	stack_op_rm binary operator application
$\oplus(m'[STP], m'[STP + 1]) \rightarrow PC := L$	bc_stack floating-point compare and conditional branch

Figure 5.5: Tiles for stack machines: The tiles refer to the stack as a memory space  $m'$ . The *stack-top pointer* STP, which points to the top of the stack. The STP must also be identified along with the tileset.

rounding mode or 1-bit value may be described by an expression  $\hat{e}$ . Figure 5.3 lists the shapes of computational tiles.

- A *control-transfer tile* changes the flow of control. Shapes include conditional and unconditional branches, and indirect branches: Figure 5.4 lists the shapes of control-transfer tiles.

Some machines, notably the Pentium floating-point unit, are not register machines but stack machines. The classification above is not useful in those cases because there are few operations that can work with an arbitrary register  $r$ . Instead, stack machines use a separate set of tiles designed to capture the types of computation implemented by a stack machine. Figure 5.5 lists the tiles for a stack machine.

## 5.4 Specifying the tiler

What distinguishes our work from previous work on code generation by tree-covering is that in our compiler, every back end uses the same set of tiles, no matter what the target machine. The tiler uses a simple maximal-munch strategy to reduce any well-typed RTL to a sequence of tiles.<sup>2</sup> For example, given the memory-to-memory move

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

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

```

t1 := 12           // load-immediate tile
t2 := ESP + t1  // binop tile
t3 := m[t2]     // load tile
t4 := 8           // load-immediate tile
t5 := ESP + t4  // binop tile
m[t5] := t3      // store tile

```

This example demonstrates how the tiler expands the code:<sup>3</sup> a single RTL is expanded into many simpler RTLs. The expansion factor is large because the tiles are small: each tile contains at most one RTL operator. The expanded code may look horribly inefficient, but in practice it is ideal for peephole optimization and other optimizations that take place after instruction selection (Davidson and Fraser 1980, 1984; Benitez and Davidson 1994). For an example of how an automatically generated expander for the x86 tiles a slightly more complicated instruction, see Figure 5.6 on page 62.

To specify the tiler, we use inference rules. The inference rules present the algorithm in a manner that is easy to understand, and they clearly communicate the division of labor between the tiler and the tileset. Because these rules are used to specify an algorithm, not to prove theorems, we take some liberties, in particular with our treatment of freshness: some of the rules declare that a temporary is *fresh*, with the understanding that the compiler can generate an infinite supply of temporaries.

The main judgment form implemented by the tiler is  $G \stackrel{\mathcal{L}}{\subseteq} G'$ , which states that the graph  $G$  implements the graph  $G'$ , but  $G$  may also overwrite any location in the set  $\mathcal{L}$ . In practice, the set  $\mathcal{L}$  should contain only scratch registers, which cannot be used or defined by the source program. The tiler's

<sup>2</sup>This step relies on the widener (Section 3.1) to ensure that every subexpression is the same width as a machine register.

<sup>3</sup>In Davidson's terminology, the tiler and the tileset cooperate to accomplish the work of a *code expander*.

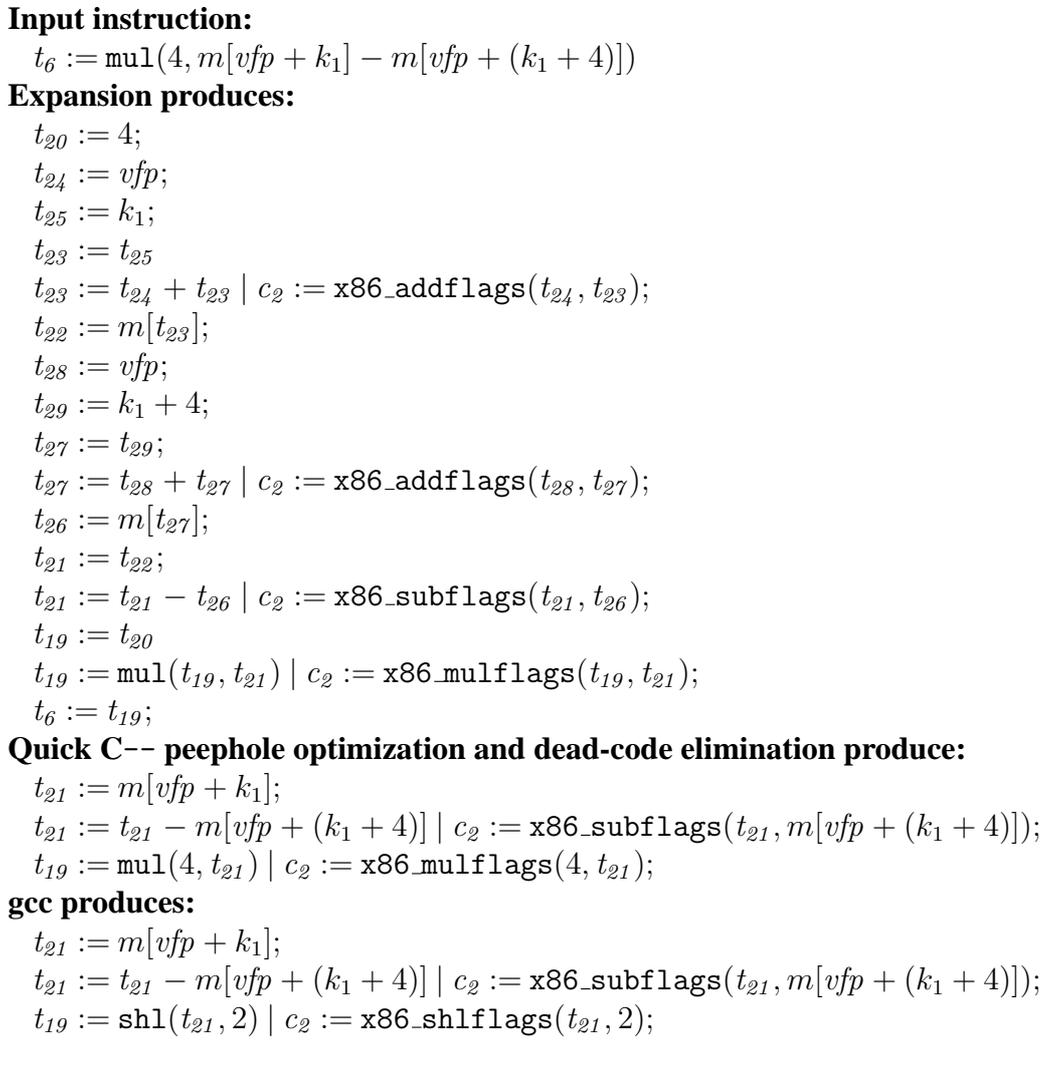


Figure 5.6: An example of expansion and peephole optimization: The input instruction fetches two values from the stack, subtracts them, and multiplies the result by 4. The expansion and first set of optimized code are produced by an automatically generated x86 back end in the Quick C-- compiler. The final set of optimized code shows the optimization that gcc performs, using our notation for RTLs. gcc's optimizer produces better code than Quick C-- by replacing the multiplication with a shift instruction; this optimization has not yet been implemented in Quick C--.

job is to find a graph  $G$  that implements the input graph  $G'$  and is composed entirely of tiles;  $G'$  is the original program IR and  $G$  is the tiling. To reduce the size of the intermediate code, the tiler includes some redundant rules, which introduce improvements that could be left to the optimizer. We note each redundant rule by adding an asterisk(\*) at the end of the rule's name.

The tiler relies on the machine-dependent tileset to implement the tiles described above. We use the following judgment form for each machine-dependent tile:  $G \stackrel{\mathcal{L}}{\subseteq}_M G'$ , which states that the graph  $G$  uses only RTLs implementable by instructions on the machine  $M$  to implement the graph  $G'$ . The graph  $G$  may also overwrite any location in the set  $\mathcal{L}$ .

Our rules use the following notation to denote that two sets are disjoint:  $S \parallel S' \triangleq S \cap S' = \emptyset$

The main judgments in the tiler compute the value of an expression  $e$  and store the result in either a temporary or an arbitrary location.

$$\boxed{G \stackrel{\mathcal{L}}{\subseteq} t := e \\ l := e}$$

We use a separate judgment form for an RTL that stores its result in a *weird temporary*. A weird temporary holds a value in its lowest bits, and the high bits may be the sign-extension or zero-extension of the value, or they may be garbage.

$$\boxed{G \stackrel{\mathcal{L}}{\subseteq} \hat{l} := \hat{e}}$$

Our final judgment form is used to explain how the tiler handles control flow:

$$\boxed{G \stackrel{\mathcal{L}}{\subseteq} G'}$$

I present these rules in groups according to the judgment in the premise, rather than the standard technique of grouping rules according to the judgment in the conclusion. The result is that the rules are grouped according to purpose: rules that move data are presented in the first section, rules that compute operator applications are presented in the second section, and rules that implement control flow are presented in the third section.

### 5.4.1 Data movement

The data-movement rules between locations of the same width are straightforward.

$$\boxed{G \stackrel{\mathcal{L}}{\subseteq}_M l_1 := l_2}$$

$$\frac{G \stackrel{\mathcal{L}}{\subseteq}_M r_1 := r_2}{G \stackrel{\mathcal{L}}{\subseteq} r_1 := r_2} \quad (\text{MOVE RR})$$

$$\frac{t' \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} t' := e \quad G' \stackrel{\mathcal{L}'}{\subseteq}_M t := m[t_2] \quad \mathcal{L} \parallel \{m[e]\}}{G; G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} t := m[e]} \quad (\text{LOAD})$$

$$\frac{t \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} t := e \quad G' \stackrel{\mathcal{L}'}{\subseteq}_M m[t] := r \quad \mathcal{L} \parallel \{r\}}{G; G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} m[e] := r} \quad (\text{STORE R}^*)$$

The STOREE rule below calls the tiler recursively to compute an expression  $e_2$  into a temporary, then uses a data-movement instruction to store the temporary to memory. The previous rule STORER\* is a special case of STOREE that generates slightly more efficient code to store a register to memory.

$$\frac{t_1, t_2 \text{ fresh} \quad \mathcal{L}_2 \parallel \text{uses}(e_1) \quad G_1 \stackrel{\mathcal{L}_1}{\subseteq} t_1 := e_1 \quad G_2 \stackrel{\mathcal{L}_2}{\subseteq} t_2 := e_2 \quad G \stackrel{\mathcal{L}}{\subseteq}_M m[t_1] := t_2}{G_2; G_1; G \stackrel{\mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}}{\subseteq} m[e_1] := e_2} \quad (\text{STOREE})$$

The BLOCKCOPY rule copies data from one memory location to another. This rule is not redundant with the other data-movement rules because the width of the data moved by a block copy can be any multiple of the machine byte width, whereas the LOAD and STORE rules can only move data that is the width of a register.

$$\frac{t_1, t_2 \text{ fresh} \quad \mathcal{L}_1 \parallel \{m[e_2]\} \quad \mathcal{L}_2 \parallel \text{uses}(e_1) \cup \{m[e_2]\} \quad G_1 \stackrel{\mathcal{L}_1}{\subseteq} t_1 := e_1 \quad G_2 \stackrel{\mathcal{L}_2}{\subseteq} t_2 := e_2 \quad G \stackrel{\mathcal{L}}{\subseteq}_M m[t_1] := m[t_2]}{G_2; G_1; G \stackrel{\mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}}{\subseteq} m[e_1] := m[e_2]} \quad (\text{BLOCKCOPY})$$

The tiler also uses a rule for loading a constant into a temporary, which immediately uses the machine-dependent tileset:

$$\boxed{G \stackrel{\mathcal{L}}{\subseteq}_M l_1 := k}$$

$$\frac{G \stackrel{\mathcal{L}}{\subseteq}_M t := k}{G \stackrel{\mathcal{L}}{\subseteq} t := k} \quad (\text{LI})$$

If we need to move a value from a hardware register that is narrower than the destination, we sign-extend or zero-extend the value in the register. We can also store a value from the low bits of a temporary into a narrow hardware register.

$$\boxed{\begin{array}{l} G \stackrel{\mathcal{L}}{\subseteq}_M l_1 := \text{sx}(l_2) \\ l_1 := \text{zx}(l_2) \\ l_1 := \text{lobits}(l_2) \\ l_1 := \text{lobits}(l \gg_l k) \end{array}}$$

$$\frac{G \stackrel{\mathcal{L}}{\subseteq}_M t := \text{sx}(r)}{G \stackrel{\mathcal{L}}{\subseteq} t := \text{sx}(r)} \quad (\text{HWGETSX})$$

$$\frac{G \stackrel{\mathcal{L}}{\subseteq}_M t := \text{zx}(r)}{G \stackrel{\mathcal{L}}{\subseteq} t := \text{zx}(r)} \quad (\text{HWGETZX})$$

$$\frac{t \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} t := e \quad G' \stackrel{\mathcal{L}'}{\subseteq}_M r := \text{lobits}_n(t)}{G; G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} r := \text{lobits}_n(e)} \quad (\text{HWSET})$$

We can load a sign-extended or zero-extended value from a narrow location in memory into a temporary.

$$\frac{t' \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} t' := e \quad G' \stackrel{\mathcal{L}'}{\subseteq}_M t := \text{sx}(m[t']) \quad \mathcal{L} \parallel \{m[e]\}}{G; G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} t := \text{sx}(m[e])} \quad (\text{SXLOAD})$$

$$\frac{t' \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} t' := e \quad G' \stackrel{\mathcal{L}'}{\subseteq}_M t := \text{zx}(m[t']) \quad \mathcal{L} \parallel \{m[e]\}}{G; G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} t := \text{zx}(m[e])} \quad (\text{ZXLOAD})$$

The following rules allow us to tile stores and fetches from narrow locations. These rules are most frequently used for compiling accesses to bit fields in C structs.

$$\frac{t_1, t_2 \text{ fresh} \quad \mathcal{L}_2 \parallel \text{uses}(e_1) \quad G_1 \stackrel{\mathcal{L}_1}{\subseteq} t_1 := e_1 \quad G_2 \stackrel{\mathcal{L}_2}{\subseteq} t_2 := e_2 \quad G \stackrel{\mathcal{L}}{\subseteq}_M m[t_1] := \text{lobits}_n(t_2)}{G_2; G_1; G \stackrel{\mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}}{\subseteq} m[e_1] := \text{lobits}_n(e_2)} \quad (\text{LOSTORE})$$

$$\frac{t' \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} t' := e \quad G' \stackrel{\mathcal{L}'}{\subseteq}_M t := \text{lobits}_n(t')}{G; G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} t := \text{lobits}_n(e)} \quad (\text{LOBITSEXTRACT})$$

$$\frac{t' \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} t' := e \quad G' \stackrel{\mathcal{L}'}{\subseteq}_M t := \text{lobits}_n(t' \gg_l lo)}{G; G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} t := \text{lobits}_n(e \gg_l lo)} \quad (\text{MIDBITSEXTRACT})$$

Recall that the notation  $l@lo:n$  describes an  $n$ -bit slice of the location  $l$ , starting at the bit  $lo$ .

We close the data-movement section with three rules that allow us to compute the value of an expression in a temporary, then move the result to another location. These rules are essential to avoid the need for movement tiles that store their results into arbitrary locations.

$$\frac{t' \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} t' := e \quad G' \stackrel{\mathcal{L}'}{\subseteq}_M t := t'}{G; G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} t := e} \quad (\text{MOVETE})$$

$$\frac{t \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} t := e \quad G' \stackrel{\mathcal{L}'}{\subseteq}_M r := t}{G; G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} r := e} \quad (\text{MOVE RE})$$

$$\frac{t \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} t := e \quad G' \stackrel{\mathcal{L}'}{\subseteq} l := t}{G; G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} l := e} \quad (\text{TOTEMP})$$

### 5.4.2 Computation

The tiler distinguishes the different types of computation by inspecting the operator's type, as described in Section 5.2. The first two rules handle unary and binary operators, which return a value that is the same width as the arguments.

$$\boxed{G \stackrel{\mathcal{L}}{\subseteq}_M l_1 := \oplus(l_2) \quad l_1 := \oplus(l_2, l_3)}$$

$$\begin{array}{c}
t' \text{ fresh} \quad \oplus : \forall n.n \text{ bits} \rightarrow n \text{ bits} \\
\frac{G \stackrel{\mathcal{L}}{\subseteq} t' := e \quad G' \stackrel{\mathcal{L}'}{\subseteq}_M t := \oplus(t')}{G; G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} t := \oplus(e)} \quad (\text{UNOP})
\end{array}$$

$$\begin{array}{c}
t_1, t_2 \text{ fresh} \quad \oplus : \forall n.n \text{ bits} \times n \text{ bits} \rightarrow n \text{ bits} \quad \mathcal{L}_1 \parallel \text{uses}(e_2) \\
\frac{G_1 \stackrel{\mathcal{L}_1}{\subseteq} t_1 := e_1 \quad G_2 \stackrel{\mathcal{L}_2}{\subseteq} t_2 := e_2 \quad G \stackrel{\mathcal{L}}{\subseteq}_M t := \oplus(t_1, t_2)}{G_1; G_2; G \stackrel{\mathcal{L} \cup \mathcal{L}_1 \cup \mathcal{L}_2}{\subseteq} t := \oplus(e_1, e_2)} \quad (\text{BINOP})
\end{array}$$

In the rule UNRM, a floating-point operator takes an additional 2-bit rounding mode as an argument. If the actual expression passed to the tile is a weird-value temporary  $\hat{t}$ , then the expression passed to the tile is a fetch of  $t$ , with the understanding that only the low 2 bits are guaranteed to have a meaningful value.

$$\begin{array}{c}
G \stackrel{\mathcal{L}}{\subseteq}_M l_1 := \oplus(l_2, \hat{l}_3) \\
l_1 := \oplus(l_2, l_3, \hat{l}_4)
\end{array}$$

$$\begin{array}{c}
t_1, \hat{t}_2 \text{ fresh} \quad \oplus : \forall n.n \text{ bits} \times 2 \text{ bits} \rightarrow n \text{ bits} \quad \mathcal{L}_1 \parallel \text{uses}(\hat{e}_2) \\
\frac{G_1 \stackrel{\mathcal{L}_1}{\subseteq} t_1 := e_1 \quad G_2 \stackrel{\mathcal{L}_2}{\subseteq} \hat{t}_2 := \hat{e}_2 \quad G \stackrel{\mathcal{L}}{\subseteq}_M t := \oplus(t_1, \hat{t}_2)}{G_1; G_2; G \stackrel{\mathcal{L} \cup \mathcal{L}_1 \cup \mathcal{L}_2}{\subseteq} t := \oplus(e_1, \hat{e}_2)} \quad (\text{UNRM})
\end{array}$$

$$\begin{array}{c}
t_1, t_2, \hat{t}_3 \text{ fresh} \quad \oplus : \forall n.n \text{ bits} \times n \text{ bits} \times 2 \text{ bits} \rightarrow n \text{ bits} \\
G_1 \stackrel{\mathcal{L}_1}{\subseteq} t_1 := e_1 \quad G_2 \stackrel{\mathcal{L}_2}{\subseteq} t_2 := e_2 \\
G_3 \stackrel{\mathcal{L}_3}{\subseteq} \hat{t}_3 := \hat{e}_3 \quad G \stackrel{\mathcal{L}}{\subseteq}_M t := \oplus(t_1, t_2, \hat{t}_3) \\
\mathcal{L}_1 \parallel \text{uses}(e_2) \cup \text{uses}(\hat{e}_3) \quad \mathcal{L}_2 \parallel \text{uses}(\hat{e}_3) \\
\frac{G_1; G_2; G_3; G \stackrel{\mathcal{L} \cup \mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}_3}{\subseteq} t := \oplus(e_1, e_2, \hat{e}_3)}{\quad} \quad (\text{BINRM})
\end{array}$$

The add-with-carry and subtract-with-borrow operators take three arguments, with the 3rd argument being one bit wide.

$$\begin{array}{l}
t_1, t_2, \hat{t}_3 \text{ fresh} \quad \oplus : \forall n. n \text{ bits} \times n \text{ bits} \times 1 \text{ bit} \rightarrow n \text{ bits} \\
G_1 \stackrel{\mathcal{L}_1}{\subseteq} t_1 := e_1 \quad G_2 \stackrel{\mathcal{L}_2}{\subseteq} t_2 := e_2 \quad G_3 \stackrel{\mathcal{L}_3}{\subseteq} \hat{t}_3 := \hat{e}_3 \\
G \stackrel{\mathcal{L}}{\subseteq}_M t := \oplus(t_1, t_2, \hat{t}_3) \quad \mathcal{L}_1 \parallel \text{uses}(e_2) \cup \text{uses}(\hat{e}_3) \quad \mathcal{L}_2 \parallel \text{uses}(\hat{e}_3) \\
\hline
G_1; G_2; G_3; G \stackrel{\mathcal{L} \cup \mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}_3}{\subseteq} t := \oplus(e_1, e_2, \hat{e}_3)
\end{array} \quad (\text{WRDOP})$$

The carry and borrow operators take three arguments, with the third argument being one bit wide; the result bit is extended and stored in a temporary.

$$\begin{array}{l}
G \stackrel{\mathcal{L}}{\subseteq}_M \hat{l}_1 := \mathbf{sx}(\oplus(l_2, l_3, \hat{l}_4)) \\
\hat{l}_1 := \mathbf{zx}(\oplus(l_2, l_3, \hat{l}_4))
\end{array}$$

$$\begin{array}{l}
t_1, t_2, \hat{t}_3 \text{ fresh} \quad \oplus : \forall n. n \text{ bits} \times n \text{ bits} \times 1 \text{ bit} \rightarrow 1 \text{ bit} \\
G_1 \stackrel{\mathcal{L}_1}{\subseteq} t_1 := e_1 \quad G_2 \stackrel{\mathcal{L}_2}{\subseteq} t_2 := e_2 \quad G_3 \stackrel{\mathcal{L}_3}{\subseteq} \hat{t}_3 := \hat{e}_3 \\
G \stackrel{\mathcal{L}}{\subseteq}_M \hat{t} := \mathbf{sx}(\oplus(t_1, t_2, \hat{t}_3)) \\
\mathcal{L}_1 \parallel \text{uses}(e_2) \cup \text{uses}(\hat{e}_3) \quad \mathcal{L}_2 \parallel \text{uses}(\hat{e}_3) \\
\hline
G_1; G_2; G_3; G \stackrel{\mathcal{L} \cup \mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}_3}{\subseteq} \hat{t} := \mathbf{sx}(\oplus(e_1, e_2, \hat{e}_3))
\end{array} \quad (\text{WRDROPSX})$$

$$\begin{array}{l}
t_1, t_2, \hat{t}_3 \text{ fresh} \quad \oplus : \forall n. n \text{ bits} \times n \text{ bits} \times 1 \text{ bit} \rightarrow 1 \text{ bit} \\
G_1 \stackrel{\mathcal{L}_1}{\subseteq} t_1 := e_1 \quad G_2 \stackrel{\mathcal{L}_2}{\subseteq} t_2 := e_2 \quad G_3 \stackrel{\mathcal{L}_3}{\subseteq} \hat{t}_3 := \hat{e}_3 \\
G \stackrel{\mathcal{L}}{\subseteq}_M \hat{t} := \mathbf{zx}(\oplus(t_1, t_2, \hat{t}_3)) \\
\mathcal{L}_1 \parallel \text{uses}(e_2) \cup \text{uses}(\hat{e}_3) \quad \mathcal{L}_2 \parallel \text{uses}(\hat{e}_3) \\
\hline
G_1; G_2; G_3; G \stackrel{\mathcal{L} \cup \mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}_3}{\subseteq} \hat{t} := \mathbf{zx}(\oplus(e_1, e_2, \hat{e}_3))
\end{array} \quad (\text{WRDROPSX})$$

Many source languages provide multiplication with an operator of the type  $32 \text{ bits} \times 32 \text{ bits} \rightarrow 32 \text{ bits}$ . But most machines provide a multiplication operation that produces a result that is twice the width of its operands:  $32 \text{ bits} \times 32 \text{ bits} \rightarrow 64 \text{ bits}$ . Of course, the result must be placed in a location that is wide enough to hold it.

To facilitate the definition of the DBLOP rule, we define functions to compute the addresses where the lower half and the upper half of a  $2n \text{ bit}$  value can be stored. Of course, the location depends on the aggregation order (little-endian or big-endian) of the target machine  $m$ :

$$\begin{aligned} \text{loaddr}(m, e, n) &= \mathbf{if} \text{ littleEndian}(m) \mathbf{then} e \mathbf{else} e + (n / \text{memsize}(m)) \\ \text{hiaddr}(m, e, n) &= \mathbf{if} \text{ littleEndian}(m) \mathbf{then} e + (n / \text{memsize}(m)) \mathbf{else} e \end{aligned}$$

$$G \stackrel{\mathcal{L}}{\subseteq}_M l_1, l_2 := \oplus(l_3, l_4)$$

$$\begin{array}{l} t_1, t_2, t_{lo}, t_{hi} \text{ fresh} \quad \otimes : \forall n. n \text{ bits} \times n \text{ bits} \rightarrow 2n \text{ bits} \\ G_1 \stackrel{\mathcal{L}_1}{\subseteq} t_1 := e_1 \quad G_2 \stackrel{\mathcal{L}_2}{\subseteq} t_2 := e_2 \quad G_3 \stackrel{\mathcal{L}_3}{\subseteq}_M t_{hi}, t_{lo} := \otimes(t_1, t_2) \\ G_4 \stackrel{\mathcal{L}}{\subseteq} m[\text{loaddr}(m, e, n)] := t_{lo} \mid m[\text{hiaddr}(m, e, n)] := t_{hi} \\ \mathcal{L}_1 \parallel \text{uses}(e_2) \cup \text{uses}(e) \quad \mathcal{L}_2 \parallel \text{uses}(e) \quad \mathcal{L}_3 \parallel \text{uses}(e) \\ \hline G_1; G_2; G_3; G \stackrel{\mathcal{L} \cup \mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}_3}{\subseteq} m[e] := \otimes(e_1, e_2) \end{array} \quad (\text{DBLOP})$$

### 5.4.3 Control flow

The tiler handles unconditional branches and conditional branches by directly using tiles in the machine-dependent tileset.

The rules for branching to labels and indirect through registers are straightforward.

$$G \stackrel{\mathcal{L}}{\subseteq}_M \mathbf{goto} L$$

$$\frac{G \stackrel{\mathcal{L}}{\subseteq}_M \mathbf{goto} L}{G \stackrel{\mathcal{L}}{\subseteq} \mathbf{goto} L} \quad (\text{BRANCHL})$$

$$\frac{t \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} t := e \quad G' \stackrel{\mathcal{L}'}{\subseteq}_M \mathbf{goto} t}{G; G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} \mathbf{goto} e} \quad (\text{BRANCHR})$$

Aside from the logical not operator, which we handle below with rule CONDBNOT, the only operators that return Boolean values are binary operators. To test a 1-bit predicate register  $r$ , our compiler phrases the test in terms of a binary comparison:  $\text{not}(r = 0)$ .

$$G \stackrel{\mathcal{L}}{\subseteq}_M \mathbf{if} \oplus(l_1, l_2) \mathbf{then} \mathbf{goto} L_T \mathbf{else} \mathbf{goto} L_F$$

$$\begin{array}{c}
t_1, t_2 \text{ fresh} \quad \oplus : \forall n. n \text{ bits} \times n \text{ bits} \rightarrow \text{bool} \\
G_1 \stackrel{\mathcal{L}_1}{\subseteq} t_1 := e_1 \quad G_2 \stackrel{\mathcal{L}_2}{\subseteq} t_2 := e_2 \quad \mathcal{L}_1 \parallel \text{uses}(e_2) \\
G \stackrel{\mathcal{L}}{\subseteq}_M \text{if } \oplus(t_1, t_2) \text{ then goto } L_T \text{ else goto } L_F \\
\hline
G_1; G_2; G \stackrel{\mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}}{\subseteq} \text{if } \oplus(e_1, e_2) \text{ then goto } L_T \text{ else goto } L_F \\
\text{(CONDB)}
\end{array}$$

If the condition is either a boolean value or the result of applying a boolean operator, the tiler compiles the conditional branch into equivalent control flow.

$$\frac{G \stackrel{\mathcal{L}}{\subseteq}_M \text{goto } L_T}{G \stackrel{\mathcal{L}}{\subseteq} \text{if } \text{true} \text{ then goto } L_T \text{ else goto } L_F} \quad \text{(CONDBT)}$$

$$\frac{G \stackrel{\mathcal{L}}{\subseteq}_M \text{goto } L_F}{G \stackrel{\mathcal{L}}{\subseteq} \text{if } \text{false} \text{ then goto } L_T \text{ else goto } L_F} \quad \text{(CONDBF)}$$

$$\frac{G \stackrel{\mathcal{L}}{\subseteq} \text{if } e \text{ then goto } L_F \text{ else goto } L_T}{G \stackrel{\mathcal{L}}{\subseteq} \text{if } \text{not}(e) \text{ then goto } L_T \text{ else goto } L_F} \quad \text{(CONDBNOT)}$$

$$\begin{array}{c}
L \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} \text{if } e_1 \text{ then goto } L \text{ else goto } L_F \\
G' \stackrel{\mathcal{L}'}{\subseteq} \text{if } e_2 \text{ then goto } L_T \text{ else goto } L_F \quad \mathcal{L} \parallel \text{uses}(e_2) \\
\hline
G; L : G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} \text{if } \text{conjoin}(e_1, e_2) \text{ then goto } L_T \text{ else goto } L_F \\
\text{(CONDBCONJOIN)}
\end{array}$$

$$\begin{array}{c}
L \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} \text{if } e_1 \text{ then goto } L_T \text{ else goto } L \\
G' \stackrel{\mathcal{L}'}{\subseteq} \text{if } e_2 \text{ then goto } L_T \text{ else goto } L_F \quad \mathcal{L} \parallel \text{uses}(e_2) \\
\hline
G; L : G' \stackrel{\mathcal{L} \cup \mathcal{L}'}{\subseteq} \text{if } \text{disjoin}(e_1, e_2) \text{ then goto } L_T \text{ else goto } L_F \\
\text{(CONDBDISJOIN)}
\end{array}$$

#### 5.4.4 Graph sequences and parallel assignments

The previous sections have shown how the tiler reduces a single assignment or a control-flow statement to tiles. The two types of graphs we have

not yet considered are sequences of graphs and RTLs with parallel assignments.

A sequence of graphs can be tiled one after the other. A precondition ensures that assignments to extra locations in  $\mathcal{L}_1$  and  $\mathcal{L}_2$  cannot change the observable semantics of the sequence.

$$\frac{G_1 \stackrel{\mathcal{L}_1}{\subseteq} G'_1 \quad G_2 \stackrel{\mathcal{L}_2}{\subseteq} G'_2 \quad \mathcal{L}_1 \parallel \text{uses}(G'_2) \quad \mathcal{L}_2 \parallel \text{defs}(G'_1) \setminus \text{defs}(G'_2)}{G_1; G_2 \stackrel{\mathcal{L}_1 \cup \mathcal{L}_2}{\subseteq} G'_1; G'_2} \quad (\text{SEQ2})$$

For an RTL with multiple parallel assignments, we want to execute each assignment in sequence, but it is important to make sure that executing one assignment does not change the semantics of the rest. If we can prove that an assignment does not modify locations that are observed by other assignments, then we can execute that assignment first:

$$\frac{G_1 \stackrel{\mathcal{L}_1}{\subseteq} l_1 := e_1 \quad \forall i > 1 : \mathcal{L}_1 \cup \{l_1\} \parallel \text{uses}(e_i) \quad G_2 \stackrel{\mathcal{L}_2}{\subseteq} l_2 := e_2 \mid \cdots \mid l_n := e_n \quad \mathcal{L}_2 \parallel \{l_1\}}{G_1; G_2 \stackrel{\mathcal{L}_1 \cup \mathcal{L}_2}{\subseteq} l_1 := e_1 \mid \cdots \mid l_n := e_n} \quad (\text{PAR})$$

We assume that the parallel assignments are reordered as necessary before applying rule PAR.

If every assignment modifies a location that is used in another assignment, then we cannot execute any assignment before the others. Instead, we break the cycle of dependencies by splitting an assignment  $l_1 := e_1$  into two parts: one assignment to store the expression  $e_1$  in a fresh temporary and one assignment to copy the value to its intended location.

$$\frac{t \text{ fresh} \quad G \stackrel{\mathcal{L}}{\subseteq} t := e_1 \mid l_2 := e_2 \mid \cdots \mid l_n := e_n ; l_1 := t}{G \stackrel{\mathcal{L}}{\subseteq} l_1 := e_1 \mid \cdots \mid l_n := e_n} \quad (\text{BREAKCYCLE})$$

By introducing a fresh temporary, we ensure that the cycle of dependencies is broken, and we will be able to make progress by applying rule PAR.

## 5.5 Possible extensions

A weakness of the tiler's design is that there is no way to make machine-dependent decisions about how to multiply and divide by compile-time con-

stants. One way to solve this problem would be to define new tiles for multiplying and dividing by constants. The tiler could then use these tiles at every opportunity. Of course, the tileset would have to provide implementations of the tiles for multiplication or division not only for constants that could be handled efficiently on the machine, but also for a default case that could handle any constant. The default case could be implemented in the same manner as the non-constant multiplication or division tile. To find the efficient implementations of multiplication or division, we can rely on the same techniques used to find implementations of any operator (see Chapter 6).

## 5.6 Properties of the tiler

The tiler translates an input graph to a new graph that uses only RTLs that are implementable on the target machine. It satisfies two properties:

- *Soundness*: If  $G \stackrel{\mathcal{L}}{\subseteq} G'$ , then  $G$  is equivalent to  $G'$ , modulo assignments to locations in  $\mathcal{L}$ .
- *Machine invariant*: If  $G \stackrel{\mathcal{L}}{\subseteq} G'$ , then  $G$  contains only RTLs implementable on the target machine.
- *Completeness*: If  $G'$  is well formed, then there exist a  $G$  and an  $\mathcal{L}$  such that  $G \stackrel{\mathcal{L}}{\subseteq} G'$ . A well-formed graph uses operators only at the word size of the target machine.<sup>4</sup>

The first two properties follow by inspection of the rules under a straightforward semantics of graphs. The third problem is much harder. The proof of completeness is outside the scope of this dissertation, but it boils down to checking the coverage of our inference rules. This coverage check is the same problem faced by the author of any instruction selector, except that they must check the property for each back end, whereas we require only a single proof on the tiler. The basic proof technique is to show that the inference rules satisfy a canonical forms lemma. The canonical forms of RTLs are defined by the expansion tiles, so our job is to show that the tiler can reduce any well-formed graph to tiles.

The proof proceeds by induction over the shape of the graph. We have already seen how sequences and parallel assignments are recursively reduced to individual graphs and single assignments. The bulk of the proof

<sup>4</sup>The restriction on the widths of operators can be relaxed to operate on values smaller than a machine word (Redwine and Ramsey 2004).

must demonstrate that any single assignment is reduced to tiles. There are two main inductive cases: the guard and the expression computed by the assignment. The induction on guards is guided by types: because Boolean expressions are distinct from bit vectors of width 1, the class of well-formed guards is constrained by the type system. Similarly, the induction on expressions is guided by the types of the operators: we must show that the inference rules cover all the type schemas of our operators. The unanswered questions lie with the width-changing operations: we must show that well-formed, nested sign-extension and bit-extraction expressions can be tiled, even though there may not be a temporary where we can store the intermediate value.

## 5.7 Obligations of the machine-dependent tileset

The tiler depends on the tileset to provide a machine-specific implementation of each tile; we refer to the collection of tiles that must be implemented as the tileset interface. We can find each of the tiles in the inference rules that define the tiler, but for convenience, we have gathered the tileset interface here.

In most cases, the obligations of a particular tile are straightforward: the tile must use only machine instructions to implement a specified RTL. But because not all instructions can operate on values in arbitrary locations, some tiles restrict source or destination operands to a limited set of locations. It is the tiler's responsibility to ensure that the operands are moved to suitable locations before using the tile. For example, to use the tile that loads a value from memory on the *x86*, the memory address must be stored in an integer register before the tile can be used.

We express the restriction on the possible locations of an operand using machine-specific predicates. Each predicate decides whether a location is a suitable location for the tile. We refer to each predicate as a *context* because it defines the context in which a location may be used.<sup>5</sup> For example, most machines use integer registers and floating-point registers in different contexts.

Aside from the context predicates, each tile is defined in the same manner as the tiler, using the familiar judgment for expansion tiles:

$$\boxed{G \stackrel{\mathcal{L}}{\subseteq}_M G'}$$

<sup>5</sup>For the purpose of implementing the tiler, the tileset also provides functions that return a fresh location in the proper context.

As usual, we divide the tiles into three categories: data movement, computation, and control flow.

### 5.7.1 Data movement

We expect to be able to move a value between any two registers of the same width, so there are no preconditions on the MOVEM tile. But loads and stores require the address in memory to be stored in a particular context.

$$\begin{array}{r}
 \frac{}{G \stackrel{\mathcal{L}}{\subseteq}_M r := r'} \quad (\text{MOVEM}) \\
 \frac{\text{LoadAddr}_M(t')}{G' \stackrel{\mathcal{L}'}{\subseteq}_M t := m[t']} \quad (\text{LOADM}) \\
 \frac{\text{StoreAddr}_M(t)}{G' \stackrel{\mathcal{L}'}{\subseteq}_M m[t] := r} \quad (\text{STOREM}) \\
 \\
 \frac{\text{BCStoreAddr}_M(t_1) \quad \text{BCLoadAddr}_M(t_2)}{G \stackrel{\mathcal{L}}{\subseteq}_M m[t_1] := m[t_2]} \quad (\text{BLOCKCOPYM})
 \end{array}$$

The rest of the data-movement tiles proceed in a similar fashion, with the tiler making recursive calls to store memory addresses to locations in the proper contexts.

$$\begin{array}{r}
 \frac{}{G \stackrel{\mathcal{L}}{\subseteq}_M t := k} \quad (\text{LIM}) \\
 \\
 \frac{}{G \stackrel{\mathcal{L}}{\subseteq}_M t := \mathbf{sx}(r)} \quad (\text{HWGETSXM}) \\
 \\
 \frac{}{G \stackrel{\mathcal{L}}{\subseteq}_M t := \mathbf{zx}(r)} \quad (\text{HWGETZXM}) \\
 \\
 \frac{}{G' \stackrel{\mathcal{L}'}{\subseteq}_M r := \text{lobits}_n(t)} \quad (\text{HWSETM}) \\
 \\
 \frac{\text{SXLoadAddr}_M(t')}{G' \stackrel{\mathcal{L}'}{\subseteq}_M t := \mathbf{sx}(m[t'])} \quad (\text{SXLOADM}) \\
 \\
 \frac{\text{ZXLoadAddr}_M(t')}{G' \stackrel{\mathcal{L}'}{\subseteq}_M t := \mathbf{zx}(m[t'])} \quad (\text{ZXLOADM})
 \end{array}$$

$$\frac{LoStoreAddr_M(t_1)}{G \stackrel{\mathcal{L}}{\subseteq}_M m[t_1] := \text{lobits}(t_2)} \quad (\text{LOSTOREM})$$

$$\frac{}{G' \stackrel{\mathcal{L}'}{\subseteq}_M t := t@lo:w} \quad (\text{EXTRACTM})$$

## 5.7.2 Computation

Each computational tile describes the application of an operator to some operands. The preconditions of the rules check the type of the operator and restrict the context of each operand. For example, on most machines the operands of an addition tile must be integer registers, which is enforced by the context predicate. Even tiles of the same shape, such as addition and subtraction tiles, may require operands to be placed in different locations, so the context predicates are indexed according to the operator.

$$\frac{\oplus : \forall n.n \text{ bits} \rightarrow n \text{ bits} \quad Arg_M(\oplus, 1, t') \quad Res_M(\oplus, t)}{G' \stackrel{\mathcal{L}'}{\subseteq}_M t := \oplus(t')} \quad (\text{UNOPM})$$

$$\frac{\oplus : \forall n.n \text{ bits} \times n \text{ bits} \rightarrow n \text{ bits} \quad Arg_M(\oplus, 1, t_1) \quad Arg_M(\oplus, 2, t_2) \quad Res_M(\oplus, t)}{G \stackrel{\mathcal{L}}{\subseteq}_M t := \oplus(t_1, t_2)} \quad (\text{BINOPM})$$

$$\frac{\oplus : \forall n.n \text{ bits} \times 2 \text{ bits} \rightarrow n \text{ bits} \quad Arg_M(\oplus, i, t_i) \quad Res_M(\oplus, t)}{G \stackrel{\mathcal{L}}{\subseteq}_M t := \oplus(t_1, \hat{t}_2)} \quad (\text{UNRMM})$$

$$\frac{\oplus : \forall n.n \text{ bits} \times n \text{ bits} \times 2 \text{ bits} \rightarrow n \text{ bits} \quad Arg_M(\oplus, i, t_i) \quad Res_M(\oplus, t)}{G \stackrel{\mathcal{L}}{\subseteq}_M t := \oplus(t_1, t_2, \hat{t}_3)} \quad (\text{BINRMM})$$

$$\frac{\oplus : \forall n.n \text{ bits} \times n \text{ bits} \times 1 \text{ bit} \rightarrow n \text{ bits} \quad Arg_M(\oplus, i, t_i) \quad Res_M(\oplus, t)}{G \stackrel{\mathcal{L}}{\subseteq}_M t := \oplus(t_1, t_2, \hat{t}_3)} \quad (\text{WRDOPM})$$

$$\frac{\oplus : \forall n.n \text{ bits} \times n \text{ bits} \times 1 \text{ bit} \rightarrow 1 \text{ bit} \quad Arg_M(\oplus, i, t_i) \quad Res_M(\oplus, t)}{G \stackrel{\mathcal{L}}{\subseteq}_M \hat{t} := \text{ext}(\oplus(t_1, t_2, \hat{t}_3))} \quad (\text{WRDOPM})$$

$$\begin{array}{c}
\oplus : \forall n. n \text{ bits} \times n \text{ bits} \rightarrow 2n \text{ bits} \\
\frac{Arg_M(\oplus, i, t_i) \quad Res_M(\oplus, t)}{G_3 \stackrel{\mathcal{L}_3}{\subseteq}_M t_{hi}, t_{lo} := \oplus(e_1, e_2)} \quad (\text{DBLOPM})
\end{array}$$

### 5.7.3 Control flow

Control-flow tiles follow a similar pattern: an unconditional branch to a label has no restrictions, but operands of a conditional must come from a limited context.

$$\frac{}{G \stackrel{\mathcal{L}}{\subseteq}_M \text{goto } L} \quad (\text{BRANCHLM})$$

$$\frac{BranchArg_M(t)}{G' \stackrel{\mathcal{L}'}{\subseteq}_M \text{goto } t} \quad (\text{BRANCHRM})$$

$$\frac{Arg_M(\oplus, 1, t_1) \quad Arg_M(\oplus, 2, t_2)}{G \stackrel{\mathcal{L}}{\subseteq}_M \text{if } \oplus(t_1, t_2) \text{ then goto } L_T \text{ else goto } L_F} \quad (\text{CONDBM})$$

## 5.8 Tileset interface

The main result of this chapter is that the tiler reduces well-typed RTLs to a set of tiles that must be implemented in every machine-dependent tileset. Table 5.7 lists each of the functions that must be implemented in a tileset, with the type of the function and the tile it implements. For the most part, the listings are straightforward. The types of the width-changing data-movement instructions have two notable features: each width-changing movement instruction involving memory requires a width argument of type  $w$ , and the extract function takes a least-significant bit argument of type  $i$ .

Function name	Tile shape	Type
$\text{move}(r_1, r_2)$	$\triangleq r_1 := r_2$	$r \times r \rightarrow G$
$\text{load}(r_1, r_2)$	$\triangleq r_1 := m[r_2]$	$r \times r \rightarrow G$
$\text{store}(r_1, r_2)$	$\triangleq m[r_1] := r_2$	$r \times r \rightarrow G$
$\text{blockcopy}(r_1, r_2, w)$	$\triangleq m[r_1] := m[r_2]:w \text{ bits}$	$r \times r \times w \rightarrow G$
$\text{li}(r, k)$	$\triangleq r := k$	$r \times k \rightarrow G$
$\text{hwget}(r, h)$	$\triangleq r := h$	$\hat{r} \times h \rightarrow G$
$\text{hwset}(h, r)$	$\triangleq h := r$	$h \times \hat{r} \rightarrow G$
$\text{sxload}(r_1, r_2, w)$	$\triangleq r_1 := \text{sx}(m[r_2]:w \text{ bits})$	$r \times r \times w \rightarrow G$
$\text{zxload}(r_1, r_2, w)$	$\triangleq r_1 := \text{zx}(m[r_2]:w \text{ bits})$	$r \times r \times w \rightarrow G$
$\text{lostore}(r_1, r_2, w)$	$\triangleq m[r_1]:w \text{ loc} := \text{lobits}(r_2)$	$r \times r \times w \rightarrow G$
$\text{extract}(r_1, lo, r_2)$	$\triangleq r_1:w \text{ loc} := r_2@lo:w \text{ bits}$	$r \times i \times r \rightarrow G$
$\text{unop}(r_1, \oplus, r_2)$	$\triangleq r_1 := \oplus(r_2)$	$r \times \oplus \times r \rightarrow G$
$\text{binop}(r_1, \oplus, r_2, r_3)$	$\triangleq r_1 := \oplus(r_2, r_3)$	$r \times \oplus \times r \times r \rightarrow G$
$\text{unrm}(r_1, \oplus, r_2, \hat{r}_3)$	$\triangleq r_1 := \oplus(r_2, \hat{r}_3)$	$r \times \oplus \times r \times \hat{r} \rightarrow G$
$\text{binrm}(r_1, \oplus, r_2, r_3, \hat{r}_4)$	$\triangleq r_1 := \oplus(r_2, r_3, \hat{r}_4)$	$r \times \oplus \times r \times r \times \hat{r} \rightarrow G$
$\text{wrdrop}(r_1, \oplus, r_2, r_3, \hat{r}_4)$	$\triangleq r_1 := \oplus(r_2, r_3, \hat{r}_4)$	$r \times \oplus \times r \times r \times \hat{r} \rightarrow G$
$\text{wrdrop}(\hat{r}_1, \oplus, r_2, r_3, \hat{r}_4)$	$\triangleq \hat{r}_1 := \text{ext}(\oplus(r_2, r_3, \hat{r}_4))$	$\hat{r} \times \oplus \times r \times r \times \hat{r} \rightarrow G$
$\text{dblop}(r_{hi}, r_{lo}, \oplus, r_1, r_2)$	$\triangleq r_{hi}, r_{lo} := \otimes(r_1, r_2)$	$r \times r \times \oplus \times r \times r \rightarrow G$
$\text{b}(L)$	$\triangleq \text{goto } L$	$L \rightarrow G$
$\text{br}(r)$	$\triangleq \text{goto } r$	$r \rightarrow G$
$\text{bc}(r_1, \oplus, r_2, L_T)$	$\triangleq \text{if } \oplus(r_1, r_2) \text{ then goto } L_T$ $\text{else goto } L_F; L_F:$	$r \times \oplus \times r \times L \rightarrow G$
$\text{call}(L)$	$\triangleq PC := L \mid LR := PC$	$L \rightarrow G$
$\text{callr}(r)$	$\triangleq PC := r \mid LR := PC$	$r \rightarrow G$
$\text{return}()$	$\triangleq PC := LR$	$() \rightarrow G$

Table 5.7: Listing of each function provided by each machine-dependent tileset, with the type of the function and the tile shape it implements.



## Chapter 6

# Generating a tileset

The tiler depends on each back end to provide machine-dependent implementations of the tiles. We generate tiles by searching a  $\lambda$ -RTL machine description for suitable combinations of instructions. In some cases, the implementation is obvious: a tile may be implemented by a single instruction on the target machine. For example, on many machines, including the PowerPC, we can implement the tile for computing addition with a single machine instruction:

$$r_1 := r_2 + r_3$$

But on some machines, a tile cannot be implemented using a single machine instruction. In such a case, we need to find a sequence of instructions to implement the tile. For example, the PowerPC does not have an instruction that computes bitwise complement. But the PowerPC does have an instruction that implements bitwise nor:

$$r_1 := \neg(r_2 \vee r_3)$$

Using a 0-identity on the bitwise or operator,  $0 \vee x = x$ , we can simulate negation by first storing the value 0 in the argument register  $r_2$ , then using the nor instruction to compute the complement of  $r_3$ .<sup>1</sup>

$$\begin{aligned} r_2 &:= 0 \\ r_1 &:= \neg(r_2 \vee r_3) \end{aligned}$$

But this sequence also introduces an assignment to  $r_2$ , which might alter the semantics of the program by overwriting a program variable stored in

---

<sup>1</sup>By ensuring that  $r_2$  is the same register as  $r_3$ , we could achieve the same result, but the example in the text illustrates a more general problem. My algorithm finds both solutions.

$r_2$ . We can compensate for this extra assignment by saving and restoring the value in  $r_2$ . To generate an implementation of the tile that implements bitwise complement on the PowerPC, our algorithm reasons about all of these problems: how to sequence instructions, how to apply algebraic laws, and how to compensate for extra assignments.

In this chapter, we explain how our algorithm finds implementations of tiles. We first discuss the essential techniques used by the tileset generator, then explain how the algorithm works using a concrete example. Finally, we describe how the algorithm is implemented.

## 6.1 Techniques for finding tiles

One way to try to generate tiles is to take each tile and use a goal-directed search to look for a sequence of instructions that implements the tile. This is more or less what Massalin's (1987) superoptimizer does, for example (see also Granlund and Kenner 1992), and it is the same approach used by Cattell (1982) in his work on generating instruction selectors. Goal-directed search can be bounded by limiting the number of instructions one is willing to combine, but as explained in Section 7.2, we believe that the number of instructions is a poor metric. Instead, we use a *forward* search that uses two data structures:

- Our algorithm maintains a *pool* of RTLs, each of which is known to be implementable by a sequence of instructions on the target machine. More precisely, we maintain a pool of RTL schemas which are parameterized over bit-vector operands. With each RTL in the pool, we associate a sequence of instructions that implements it. We initialize the pool by expanding every nonterminal symbol in the  $\lambda$ -RTL grammar and taking the resulting set of RTLs.<sup>2</sup> Each RTL in the initial pool is associated with a sequence containing exactly one instruction.
- Our algorithm also maintains a set of unimplemented tiles. We initialize the set to include all tiles.

We specify a nondeterministic search, which proceeds by making one of two steps:

1. Using RTLs in the pool, we construct a sequence of instructions that implements a new RTL, which we then add to the pool.

---

<sup>2</sup>The set of RTLs from the machine description is pruned according to the procedure in Section 6.5 on page 102.

2. We find a way to implement a tile using an RTL from the pool; we then remove that tile from the set of unimplemented tiles.

We repeat one step or the other until there are no more unimplemented tiles. One difficulty is that there is no inherent limit to the size of the pool, so it is not obvious when to stop adding things in Step 1. Section 6.2.1 on page 90 describes how we choose RTLs to combine and how we avoid repeating Step 1 indefinitely.

Step 1 looks for a sequence of RTLs in the pool such that each RTL in the sequence (except the last) assigns to a location that is read by a later RTL. The sequence of RTLs computes an expression that is more complicated than the expressions used in the individual RTLs; we identify the expression computed by substituting forward for assigned locations.

By design, our tiles require simple expressions, not complicated ones (Table 5.7 on page 77). A sequence of RTLs that computes a complicated expression may be useful only if its expression is equivalent to a simpler expression that is useful for implementing tiles (as discussed in Section 6.5). For instance, the example at the start of this chapter showed how computing the expression  $\neg(r_1 \vee r_2)$  could be used to implement the simpler expression  $\neg r_2$ . In such cases, we add the implementation to the pool.

Although implementing tiles may sound complicated, it can be understood by considering three simple ideas:

- *Algebraic laws*: An algebraic law asserts that two expressions are equivalent. In particular, it can show that an implementation of a complicated expression may also serve as an implementation of a simpler, equivalent expression.<sup>3</sup> For example, the algebraic law  $0 \vee x = x$  is used above to find the implementation of bitwise complement on the PowerPC.
- *Compensation for extra assignments*: A sequence of instructions may have more than one assignment; to avoid introducing unwanted assignments in the program we are compiling, we have to compensate for them. In our example implementation of bitwise complement, we introduced an unwanted assignment to the register  $r_2$ .
- *Data movement*: A recurring problem is to get the value of an expression from where it is computed to where it can be used. This ability is needed to implement the data-movement tiles and also to save and

---

<sup>3</sup>An algebraic law can also show that a simpler expression can implement a more complicated expression, but in practice, this direction is not useful.

restore locations. In addition, by inserting data-movement RTLs between two RTLs that assign and read different locations, we may be able to add more useful RTLs to the pool.

I describe each of these techniques, then describe my implementation in the  $\lambda$ -RTL toolkit.

### 6.1.1 Algebraic laws and equivalence of expressions

Equivalent expressions are often syntactically identical, but in general we use algebraic laws to show equivalence. For example, the algebraic law  $0 \vee x = x$  helps show that the PowerPC's `orc` instruction can implement bitwise complement. Many laws are left or right identities of binary operators. One of the most frequently used is the right identity of addition:  $x + 0 = x$ . On many RISC machines, including PowerPC, this identity enables us to discover that the `add-immediate` instruction

$$r_1 := r_2 + k$$

can be used to implement the register-register move tile, provided we set  $k = 0$ .

We also use operator-inverse laws, such as  $\text{neg}(\text{neg}(x)) = x$ . Both inverses and identities are examples of a larger class of laws in which an expression is shown to be equivalent to a more complicated expression involving itself. Sometimes the expression can appear more than once, as in the law  $((x \gg_l n) \ll n) \vee \text{zx}(\text{lobits}_n(x)) = x$ . This law is useful on RISC machines to show that we can load a 32-bit constant into a register by a sequence of `load-upper-immediate` and `or-immediate` instructions.

Another very useful kind of law shows how to implement a single operator in terms of other operators. These laws appear to be endless, and they embody some of the kind of domain-specific knowledge presented by Warren (2002). For example, we can implement bitwise complement using integer negation and integer subtraction, or we can implement integer negation using bitwise complement and integer addition:

$$\begin{aligned} \text{com}(e) &= \text{neg}(e) - 1 \\ \text{neg}(e) &= \text{com}(e) + 1 \end{aligned}$$

Table 6.1 shows how we divide our algebraic laws into different categories: we give a representative example of each category as well as the number of laws. These laws do not represent a complete algebraic theory: they represent only a small subset of laws we have needed to develop our back ends. For a complete list of the algebraic laws I use, see Appendix C.

Category	Example	Count
Identities	$x + 0 = x$	8
Inverses	$-(-x) = x$	4
Operator implementations	$x - ((x/y) \times y) = \text{rem}(x, y)$	15
Machine-specific operators	<code>ppc_eq(ppc_signed_cmp(x, y)) = eq(x, y)</code>	75
Aggregating identities	$(e \gg_l n) \ll n \vee \text{zx}(\text{lobits}_n(e)) = e$	24

Table 6.1: Examples and counts for each category of algebraic laws on expressions. A complete list of the algebraic laws can be found in Appendix C.

The relatively large number of machine-specific algebraic laws is due to the machine-specific handling of condition codes. For each back end, we have to write algebraic laws that relate the operations that set and check the condition codes to the conditional expressions we want to check. These algebraic laws arise from the practical desire to write machine descriptions that abstract away from details like the modified bits in the condition-code registers (Section 2.4.1 on page 23). For example, the algebraic law

$$\text{ppc\_eq}(\text{ppc\_signed\_cmp}(x, y)) = \text{eq}(x, y)$$

is used to show that if one instruction sets the condition codes using the `ppc_signed_cmp` operator, then another instruction can use the `ppc_eq` operator to check the condition codes to see whether the operands were equal.

We divide the laws into different kinds only to help us explain what is going on. In practice, our algorithm treats every law in the same way: any expression that matches an algebraic law can be replaced with the equivalent expression specified by the law. In this way, we use algebraic laws to conclude that an implementation of one RTL in the RTL pool can also be used to implement an equivalent RTL.

Sometimes it is necessary to reason not only about RTLs but also about control-flow graphs. The standard example is when we have to use control flow to implement an operator. For example, we can check for overflow of division, which produces an exception on many architectures (Kane 1989), by checking both that the dividend is the minimum integer value<sup>4</sup> and that

<sup>4</sup>Note: There is no built-in value *minint* for the minimum integer: we compute the value of *minint* with a shift instruction.

the divisor is  $-1$ :

```
(if div_overflows( $x, y$ ) then goto  $L_T$  else goto  $L_F$ ) =
(if  $x = \text{minint}$  then (if  $y = -1$  then goto  $L_T$  else goto  $L_F$ )
else goto  $L_F$ )
```

Because we do not want to introduce complicated reasoning about control flow, we express the implementation as an algebraic law on a simple graph. A rewrite law  $G_1 = G_2$  for graphs is applied by attempting to find an implementation from the pool for each of the RTLs (including control-flow instructions) in  $G_1$ . If we can implement  $G_1$ , then we can conclude that the implementation also implements  $G_2$ . We currently use only five algebraic laws to rewrite graphs, but we expect to add more for implementing somewhat unusual operations such as population count (Warren 2002, Chapter 5, pp 65-74).

### 6.1.2 Compensating for extra assignments

Many RTLs will contain unwanted assignments. Assignments to condition codes are almost always unwanted, because our tiler does not use condition codes; computational tiles assign only to their result registers, and conditional-branch tiles test conditions directly. Another source of unwanted assignments is the combination of RTLs in sequence. Earlier assignments in a sequence are typically not mentioned in a tile. For example, the sequence

$$\begin{aligned} r_1 &:= 0 \\ r_3 &:= r_1 + r_2 \end{aligned}$$

is equivalent to the RTL

$$r_1 := 0 \mid r_3 := r_2$$

This RTL would be a data-movement tile if not for the unwanted assignment to  $r_1$ .

To find implementations of the tiles, we need to compensate for extra assignments in RTLs. We can compensate for an unwanted assignment to a location  $l$  in an RTL either by ensuring that the unwanted assignment cannot be observed by the rest of the program or by saving and restoring  $l$ .

An assignment to a location  $l$  is unobservable if we can guarantee that it never reaches an RTL that reads  $l$ . We provide the guarantee by making  $l$  a *fresh temporary* or a *scratch register*.

Our compiler provides an infinite supply of fresh temporaries, each of which is guaranteed to be distinct from every other location used in the program. If we use a fresh temporary on the left-hand side of an unwanted assignment, the assignment cannot be observed. In the example above, we can use temporary  $t_0$  instead of  $r_1$ :

$$\begin{aligned}t_0 &:= 0 \\ r_3 &:= t_0 + r_2\end{aligned}$$

In a similar fashion, we can use a fresh stack slot to compensate for an extra assignment to a memory location.

There's nothing magical about fresh temporaries; an infinite supply of distinct locations is a standard abstraction—to map such locations to finite hardware, we use a register allocator. But not every hardware resource should be managed by a register allocator; for a unique resource like a condition-code register, we can save and restore, or we can deploy our other strategy: scratch registers.

A back end can take any set of hardware registers and make them unnameable in source code and unavailable to the register allocator. These registers then become available as *scratch registers*. Scratch registers may be used freely within the implementation of a single tile, provided that no scratch register is live in or live out at a tile. Together, these properties guarantee that an assignment to a scratch register will not affect the observable behavior of a program. Some care is still required: within the implementation of a single tile, only one value at a time can be stored in each scratch register.

An example of a scratch register is the condition-code register. It is not exposed to source code, it cannot usefully be managed by the register allocator, and in the presence of conditional-branch instructions it would be awkward to save and restore: restore instructions would have to be inserted not only after the branch instruction but also at the branch target. Making the condition-code register a scratch register allows the tileset generator to mutate it at will.

Another example of a scratch register is the Y register on the SPARC. The Y register is used as temporary storage by some instructions, including division and multiplication, which use the Y register to store the high bits of operands and results.

### 6.1.3 Data-movement graph

The data-movement tiles must be able to move a value between any two registers, as well as between registers and memory. Because data-movement

tiles are also used to save and restore locations and to connect multiple RTLs in sequence, they play a key role in generating the tileset. In general, as we discover implementations of new data-movement tiles in Step 2, we create new opportunities to add useful RTLs to the pool in Step 1. For example, to implement the tile for `carry`, on the `x86` we first perform an addition instruction, then use a data-movement tile to move the carry-out bit from the condition-code register into a general-purpose register. We cannot find the implementation of `carry` until we have found a way to move bits from the condition-code register to a general-purpose register. To help find data-movement tiles, we have a special analysis and data structure.

To find data-movement tiles, we build a directed *data-movement graph*. Each node in the graph represents a set of locations on the machine, determined using the *location-set analysis* of Feigenbaum (2001), as modified by Dias and Ramsey (2006). This analysis identifies locations that are interchangeable for use in some subset of instructions.<sup>5</sup> For example, floating-point registers are typically interchangeable because most instructions that refer to one floating-point register can refer to any floating-point register. As another example, most machines have a set of general-purpose registers that are interchangeable for most purposes, although some registers may play special roles in call instructions or multiply instructions. The contribution in this dissertation is that we not only identify what locations are interchangeable, but we automatically find ways to move values among and between interchangeable locations. That is, we find instructions that move values from one register to another of the same kind, instructions that move values between registers of different kinds, and instructions that move values between registers and memory. On some platforms, values have to be moved indirectly, e.g., from integer registers to floating-point registers via memory. This is why we have a data-movement *graph*; to move a value from one location to another, it is sufficient that there is a path from one corresponding graph node to the other.

Ideally the 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. In practice, the data-movement graph usually lacks some edges involving fixed registers. For example, on the `x86`, only 8 bits of the 16-bit condition-code register can be moved to other locations. Luckily, these missing edges have little practical consequence because with just a few ex-

---

<sup>5</sup>The results of the interchangeability analysis also play a key role in register allocation (Smith, Ramsey, and Holloway 2004). For example, if a machine has floating-point registers, they are almost certainly *not* interchangeable with integer registers, and the register allocator must distinguish floating-point temporaries from integer temporaries.

ceptions (hwget, hwset, call, return; see Table 5.7 on page 77), tiles mention only general-purpose registers.

## 6.2 The algorithm in action

Now that we have described the basic techniques used by our algorithm, we can describe the algorithm itself. We begin by describing how we enlarge a non-empty pool; initialization of a pool is discussed in Section 6.3.3 on page 96.

Our algorithm takes an RTL in the pool and tries to sequence it with other RTLs to compute something useful. The strategy is to take an RTL that computes an expression  $e$  and use an algebraic law to show that the RTL can be sequenced with other RTLs to compute a new expression  $e'$ . For example, given a multiply-and-add instruction implemented by the graph  $G_{muladd}$ :

$$G_{muladd} \triangleq r_1 := (r_2 \times r_3) + r_4$$

we can implement an addition instruction. We show that the expression  $(r_2 \times r_3) + r_4$  can implement the addition  $r_2 + r_4$  by applying the algebraic law  $x \times 1 = x$ , where  $x = r_2$  and  $r_3 = 1$ . But we can only apply this algebraic law if we can ensure that  $r_3 = 1$ . Therefore to establish  $r_3 = 1$ , we want to sequence the addition instruction with a load-immediate, which we hope to find in the pool. We refer to the resulting sequence of instructions as  $G_{add+}$ , and we note the program point between the two instructions with an asterisk (\*):

$$\begin{array}{c} G_{add+} \triangleq r_3 := 1 \\ * \\ r_1 := (r_2 \times r_3) + r_4 \end{array}$$

Using the algebraic law, we can conclude that the graph  $G_{add+}$  implements the addition along with an extra assignment to  $r_3$ :

$$G_{add+} \equiv r_1 := r_2 + r_4 \mid r_3 := 1$$

We can compensate for the extra assignment by saving and restoring  $r_3$  to a fresh temporary  $t$ , resulting in the final implementation graph  $G_{add}$ :

$$\begin{array}{c} G_{add} \triangleq t := r_3 \\ r_3 := 1 \\ r_1 := (r_2 \times r_3) + r_4 \\ r_3 := t \end{array}$$

The effect of this sequence is to implement the addition instruction along with an assignment to the temporary  $t$ :

$$G_{add} \equiv r_1 := r_2 + r_4 \mid t := r_3$$

Because an assignment to a fresh temporary is unobservable, the sequence of instructions implements the addition instruction with no other observable assignments, so we can add the implementation to the pool.

As our example demonstrates, finding an implementation of a new RTL requires us to reason about the state of the machine. For example, at program point  $*$ , we must know that  $r_3 = 1$ . An excellent tool for this kind of reasoning is Hoare logic.

We use an extended form of Hoare logic augmented with constraints. The main judgment is of the form

$$\mathcal{C} \vdash \{P\} G \{Q \wedge \text{modifies } \mathcal{L}\}$$

which states that provided that constraints  $\mathcal{C}$  are satisfied, executing graph  $G$  in a state satisfying the precondition  $P$  results in a state satisfying the postcondition  $Q$ . The postcondition is a 2-state predicate that can refer to the machine state both before and after the graph executes; therefore, the postcondition makes sense only in a Hoare triple. To refer to a location  $l$  in the input state, a postcondition uses the notation  $\text{in } l$ ; the same location in the output state is named without any special notation.<sup>6</sup> The judgment also states the set of locations  $\mathcal{L}$  modified by the graph, which is used to keep track of extra assignments. If a location is not in the set  $\mathcal{L}$ , its value must remain unchanged. The final part of the judgment is a set of constraints  $\mathcal{C}$  on variables standing for compile-time constants (Section 3.1), location sets (Section 4.1.2), and expressions (Section 6.3.1). Each constraint specifies that a variable is equal to some expression. In our example above, when applying the algebraic law  $x \times 1 = x$ , we can use a constraint to assert that the metavariable  $x$  equates to  $r_2$ .

As examples of the judgment, we describe two of the implementations in the pool: the multiply-and-add instruction and the add instruction. Each implementation in the pool consists of three parts: a graph, the RTL implemented by the graph, and a set of locations modified by the graph. The following judgment describes the implementation of multiply-and-add:

$$\vdash \{true\} G_{muladd} \{r_1 = (\text{in } r_2 \times \text{in } r_3) + \text{in } r_4 \wedge \text{modifies } \{r_1\}\}$$

<sup>6</sup>The notation can be lifted to expressions homomorphically by applying  $\text{in}$  to each location in the expression.

Like all implementations in the pool, the graph does not require any preconditions or constraints, and the postcondition states the assignment implemented by the graph. Because an implementation in the pool is useful only if it does not have extra observable assignments, the set  $\mathcal{L}$  should contain only temporaries, scratch registers, and the locations we want the implementation to modify. The implementation of the addition instruction includes the assignment to the temporary  $t$  in the set of modified locations:

$$\vdash \{true\} G_{add} \{r_1 = \mathbf{in} r_2 + \mathbf{in} r_4 \wedge \text{modifies} \{r_1, t\}\}$$

To demonstrate how the judgment helps us reason about finding implementations of new RTLs, we walk through our example, looking at the judgments used in our algorithm. We begin by choosing the multiply-and-add instruction from the pool, which gives us the following judgment:

$$\vdash \{true\} G_{muladd} \{r_1 = (\mathbf{in} r_2 \times \mathbf{in} r_3) + \mathbf{in} r_4 \wedge \text{modifies} \{r_1\}\}$$

We can apply the algebraic law  $x \times 1 = x$  only if we can establish that  $\mathbf{in} r_2 \times \mathbf{in} r_3 = x \times 1$ . The establishment process is a bit like unification, and we can establish the equality  $x = \mathbf{in} r_2$  with a constraint. But  $\mathbf{in} r_3 = 1$  is not a constraint on metavariables: it is a precondition on the state of the machine. Using the constraint and the precondition, we can apply the algebraic law to conclude that  $\mathbf{in} r_2 \times \mathbf{in} r_3 = x$ , resulting in the following judgment:

$$x = \mathbf{in} r_2 \vdash \{r_3 = 1\} G_{muladd} \{r_1 = x + \mathbf{in} r_4 \wedge \text{modifies} \{r_1\}\}$$

But before we can use this implementation for addition, we must eliminate the precondition required by  $G_{muladd}$  by finding a graph that establishes  $r_3 = 1$ . We can use the implementation

$$\vdash \{true\} r_3 := 1 \{r_3 = 1 \wedge \text{modifies} \{r_3\}\}$$

from the pool to establish the precondition. The resulting sequence  $G_{add+}$  results in the following judgment:

$$x = \mathbf{in} r_2 \vdash \{true\} G_{add+} \{r_1 = x + \mathbf{in} r_4 \wedge \text{modifies} \{r_1, r_3\}\}$$

Then, we can eliminate the constraint  $x = \mathbf{in} r_2$  by substitution, allowing us to conclude that  $r_1 = \mathbf{in} r_2 + \mathbf{in} r_4$ :

$$\vdash \{true\} G_{add+} \{r_1 = \mathbf{in} r_2 + \mathbf{in} r_4 \wedge \text{modifies} \{r_1, r_3\}\}$$

The graph  $G_{add+}$  implements an addition instruction that stores its result in  $r_1$ , but it also makes an extra assignment to the location  $r_3$ . We can use a

pair of graphs from the pool to save and restore the location  $r_3$ , resulting in the graph  $G_{add}$  with the following judgment:

$$\vdash \{true\} G_{add} \{r_1 = \mathbf{in} r_2 + \mathbf{in} r_4 \wedge \text{modifies} \{r_1, t\}\}$$

Because the extra assignment to the temporary  $t$  is not observable, the implementation can be added to the pool.

### 6.2.1 The shape of the algorithm

We can generalize from this example to see the shape of our algorithm. We repeat the development of the previous section, introducing metavariables to describe the general case. In our example, we began with a judgment on the multiply-and-add instruction from the pool:

$$\vdash \{true\} G_{muladd} \{r_1 = (\mathbf{in} r_2 \times \mathbf{in} r_3) + \mathbf{in} r_4 \wedge \text{modifies} \{r_1\}\}$$

In general, our algorithm chooses a graph  $G_0$  that implements the assignment  $l_0 := e_0$  from the pool, satisfying a general judgment:

$$\vdash \{true\} G_0 \{l_0 = \mathbf{in} e_0 \wedge \text{modifies} \mathcal{L}\}$$

where the metavariables  $G_0$ ,  $l_0$ ,  $e_0$ , and  $\mathcal{L}$  replace the terms  $G_{muladd}$ ,  $r_1$ ,  $(\mathbf{in} r_2 \times \mathbf{in} r_3) + \mathbf{in} r_4$ , and  $\{r_1\}$ .

In our example, we picked the subexpression  $\mathbf{in} r_2 \times \mathbf{in} r_3$  to be rewritten by the algebraic law  $x \times 1 = x$ . In general, our algorithm picks a subexpression  $e$  of  $e_0$  (or possibly  $e_0$  itself); we say that  $e_0 = E[e]$ , where  $E$  represents an evaluation context containing the subexpression  $e$ . In our example, the expression chosen for  $e$  is  $\mathbf{in} r_2 \times \mathbf{in} r_3$ , and the context  $E$  is  $[\ ] + \mathbf{in} r_4$ . In cases where the context does not matter, we may use the notation  $e \succ e_0$  to choose an expression  $e$  from  $e_0$ . The binary operator  $\succ$  is intended to evoke the image of a tree, where the expression on the left is a subtree and is therefore part of the expression on the right.

Our algorithm then chooses an algebraic law of the form  $e_1 = e_2$  and attempts to find a set of constraints  $\mathcal{C}$  and instructions from the pool to establish a state in which  $e = e_1$ . In our example, we had to establish a state in which  $x \times 1 = \mathbf{in} r_2 \times \mathbf{in} r_3$ . The resulting judgment used a constraint on the variable  $x$  to establish that  $x = \mathbf{in} r_2$  and sequenced the graph with a load-immediate instruction to establish the precondition  $r_3 = 1$ . Unlike our example in the previous section, which described the process of finding the constraint and the load-immediate instruction in multiple steps, our algorithm uses a single call to an establishment procedure to find both the constraints and the graph:

$$x = \mathbf{in} r_2 \vdash \{true\} r_3 := 1 \{x \times 1 = \mathbf{in} r_2 \times \mathbf{in} r_3 \wedge \text{modifies} \{r_3\}\}$$

In general, to apply an algebraic law, we need to find a set of constraints  $\mathcal{C}$  on variables and a graph  $G$  to ensure that  $e = e_1$ . The resulting judgment has the form

$$\mathcal{C} \vdash \{true\} G \{e = e_1 \wedge \text{modifies } \mathcal{L}'\}$$

where  $\mathcal{C}$  stands for the constraint binding  $x = \mathbf{in } r_2$  in our example, and  $G$  stands for the graph  $r_3 := 1$ . We should note that, to find a sequence of instructions to satisfy the condition  $e = e_1$ , our algorithm uses only implementations that are already in the pool, which helps restrict the search space (for more details see Section 6.4 on page 101).

Having established that the algebraic law applies to an expression, we can sequence the graphs and rewrite the expression. In our example, we concluded that the expression  $\mathbf{in } r_2 \times \mathbf{in } r_3$  could be rewritten to  $x$  using the judgment

$$x = \mathbf{in } r_2 \vdash \{true\} G_{add+} \{r_1 = x + \mathbf{in } r_4 \wedge \text{modifies } \{r_1, r_3\}\}$$

In general, once we have shown that  $e = e_1$  in the state established by graph  $G$ , we can apply the algebraic law to conclude that  $e = e_2$  and rewrite the expression  $e$  to  $e_2$  in its original context:

$$\mathcal{C} \vdash \{true\} G; G_0 \{l_0 = \mathbf{in } E[e_2] \wedge \text{modifies } \mathcal{L} \cup \mathcal{L}'\}$$

If the graph  $G$  introduces extra assignments to observable locations, then we need to compensate for the extra assignments. In our example, we had to compensate for the assignment to the location  $r_3$  by introducing graphs to save and restore  $r_3$ . Unlike our example, we compensate for extra assignments before eliminating constraints. In general, our algorithm may introduce new graphs  $G_{save}$  and  $G_{restore}$  to save and restore observable locations. But if the location can be replaced with an unobservable location such as a fresh temporary, then we can add a constraint to that effect and use empty graphs in place of the save and restore graphs:

$$\mathcal{C}' \vdash \{true\} G_{save}; G; G_0; G_{restore} \{l_0 = \mathbf{in } E[e_2] \wedge \text{modifies } \mathcal{L}''\}$$

The implementation can be added to the pool only if the set of locations  $\mathcal{L}''$  does not contain any unwanted observable locations. It should be noted that we can compensate not only for extra assignments in the graph  $G$ , but also for extra assignments in  $G_0$ . For example, if  $G_0$  is an addition instruction with an extra assignment to the condition codes, we can save and restore the condition codes.

Finally, we can eliminate the constraints by substitution, leading to the final judgment in our example:

$$\vdash \{true\} G_{add} \{r_1 = \mathbf{in} r_2 + \mathbf{in} r_4 \wedge \text{modifies} \{r_1, t\}\}$$

In general, the substitution must be applied to each of the graphs, locations, and expressions, resulting in the judgment:

$$\vdash \{true\} G'_{save}; G'; G'_0; G'_{restore} \{l'_0 = \mathbf{in} E'[e'_2] \wedge \text{modifies} \mathcal{L}'''\}$$

## 6.2.2 Connecting the tileset and the tiler

We have described our algorithm for generating implementations of expansion tiles using a judgment that makes use of Hoare logic, but the tiler expects the tile to satisfy a simpler refinement relation. Fortunately, given an implementation of a tile satisfying a judgment

$$\mathcal{C} \vdash \{true\} G \{l_1 = \mathbf{in} e_1 \wedge \dots \wedge l_n = \mathbf{in} e_n \wedge \text{modifies} \mathcal{L}\}$$

it is easy to derive the corresponding refinement relation. Because the judgment in Hoare logic asserts that the graph  $G$  establishes a state in which the locations  $l_i$  hold the expressions  $\mathbf{in} e_i$ , we can say that the graph  $G$  implements the RTL consisting of the parallel composition of the assignments  $l_i := e_i$ . The single minor complication is the set of constraints. As described in Section 6.3.1, the constraints can be solved in a straightforward fashion, producing a substitution  $\hat{\mathcal{C}}$  that maps each variable in a graph to its solution. By applying the substitution to the graph  $G$  and the implemented RTL, we arrive at a judgment using the refinement relation for the tiler:

$$\hat{\mathcal{C}}(G) \stackrel{\hat{\mathcal{C}}(\mathcal{L})}{\subseteq} \hat{\mathcal{C}}(l_1 := e_1 \mid \dots \mid l_n := e_n)$$

## 6.3 The algorithm

At this point, we have explained how our algorithm adds implementations of new RTLs to the pool, but we have not actually stated the algorithm. The algorithm is given by the high-level pseudocode in Figure 6.2, which we explain in enough detail to implement the algorithm. We begin with just enough formalism to explain the precise meaning of the Hoare-logic judgment used by our algorithm, then we describe the algorithm itself.

```

1 Initialize the pool
2 repeat until no implementations are added to the pool
3   Build the data-movement graph from current contents of the pool
4   foreach element  $\vdash \{true\} G_0 \{l_0 = \mathbf{in} e_0 \wedge \text{modifies } \mathcal{L}\}$  in pool
5     foreach subexpression and context  $E[e] = e_0$ 
6       foreach law  $e_1 = e_2$ 
7 Establishment: if the pool can establish  $e = e_1$  by  $G$  under  $\mathcal{C}$  then
8         let  $G_{save}, G_{restore}$  save and restore observable locations modified by  $G, G_0$ 
9         Normalize  $\mathcal{C} \vdash \{true\} G_{save}; G; G_0; G_{restore} \{l_0 = \mathbf{in} E[e_2] \wedge \text{modifies } \mathcal{L}'\}$ 
10        to get  $\vdash \{true\} G'_{save}; G'; G'_0; G'_{restore} \{l'_0 = \mathbf{in} E'[e'_2] \wedge \text{modifies } \mathcal{L}''\}$ 
11 Pruning: if  $l'_0 = \mathbf{in} E'[e'_2]$  passes a utility test (see Section 6.5) then
12   pool  $\text{+=} \vdash \{true\} G'_{save}; G'; G'_0; G'_{restore} \{l'_0 = \mathbf{in} E'[e'_2] \wedge \text{modifies } \mathcal{L}''\}$ 

```

Figure 6.2: Pseudocode summarizing our algorithm. We highlight the establishment and pruning steps because they are the most complicated parts of the algorithm.

Variable kind	Constraint	Restrictions on form of constraint
Compile-time constant	$x_k = e$	$e$ must be a compile-time expression
Location	$x_l; l_s = l$	
Expression	$x_e = e$	

Table 6.3: There is one form of constraint for each kind of variable. Compile-time constants may only be equated with compile-time expressions. Although this restriction could be imposed syntactically, we prefer not to tailor the grammar to this purpose.

### 6.3.1 The Hoare-logic judgment

To understand how our algorithm works, we need to understand the details of the Hoare-logic judgment

$$\mathcal{C} \vdash \{P\} G \{Q \wedge \text{modifies } \mathcal{L}\}$$

In particular, we explain the constraints used by the algorithm and we give a model-theoretic explanation of the judgment.

Our algorithm generates constraints on variables standing for compile-time constants (Section 3.1), location sets (Section 4.1.2), and expressions (this section). The first two types of variables result from our analyses of the instructions in a machine description, which are described in Chapter 4; expression variables are new. An expression variable stands for an expression when we write an algebraic law such as  $x_e + 0 = x_e$ . In Figure 6.4, we extend our grammar of RTLs by adding these variables. For each type

Loc types	$\tau_l ::= n \text{ loc}$
Exp types	$\tau_e ::= \text{bool} \mid n \text{ bits}$
Integers	$i, n$
Bool	$b ::= \text{true} \mid \text{false}$
Constants	$k ::= b \mid i \mid x_k$
Storage spaces	$s ::= 'a' \mid \dots \mid 'z'$
Single Locations	$sl ::= r \mid t \mid s[e]:\tau_l \mid sl@n:\tau_l$
Location Sets	$ls ::= \{sl, \dots, sl\}$
Locations	$l ::= sl \mid x_l : ls$
Expressions	$e ::= k \mid l \mid \oplus(e_1, \dots, e_n) \mid x_e \mid \mathbf{in} \ l$
Guarded Assignment	$g ::= e \rightarrow l := e$
RTLs	$r ::= g_1 \mid \dots \mid g_n$
Graph	$G ::= \text{nop} \mid r \mid G; G$

Figure 6.4: The extended RTL grammar, with constant variables, location variables, expression variables, and the **in** qualifier for locations.

of variable, our algorithm generates one form of constraint, as specified in Table 6.3. We solve this limited form of constraints using a straightforward implementation of unification.

### 6.3.2 Model-theoretic account of the judgment

Because our judgment is designed to show how a graph changes the state of the machine, the model for our judgment is straightforward. But before we can define the model, we have to describe the elements of the Hoare triple.

A graph  $G$  is a function from states to states. A precondition  $P$  is a predicate on the initial state  $\sigma$  in which the graph is executed. The postcondition  $Q$ , on the other hand, is a 2-state predicate (Wing 1983; Jones 1986; Guttag and Horning 1993) that may refer to locations in both the initial state  $\sigma$  and the final state  $\sigma'$  that results from executing the graph. For a postcondition to refer to a location  $l$  in the initial state, we introduce the notation **in**  $l$  (see the extended RTL grammar in Figure 6.4). The logic used in both predicates admits only conjunction of equalities on RTL expressions.

The interpretations of graphs, preconditions, and postconditions have the following types:

$$\begin{aligned} \llbracket G \rrbracket &: \text{state} \rightarrow \text{state} \\ \llbracket P \rrbracket &: \text{state} \rightarrow \text{bool} \\ \llbracket Q \rrbracket &: \text{state} \rightarrow \text{state} \rightarrow \text{bool} \end{aligned}$$

Each of these interpretations proceeds in an intuitive fashion: the interpretation of a graph computes the final state that results from evaluating the graph  $G$  in the initial state,<sup>7</sup> the precondition is evaluated on the initial state, and the postcondition is evaluated on both the initial and final states. The only interesting cases are the interpretations of locations in the predicates:

$$\begin{aligned} \llbracket l \rrbracket \sigma_{in} &= \sigma_{in}(l) \\ \llbracket l \rrbracket (\sigma_{in}, \sigma_{out}) &= \sigma_{out}(l) \\ \llbracket \mathbf{in} \ l \rrbracket (\sigma_{in}, \sigma_{out}) &= \sigma_{in}(l) \\ \llbracket \mathit{modifies} \ \mathcal{L} \rrbracket (\sigma_{in}, \sigma_{out}) &= \forall l \notin \mathcal{L} : \sigma_{in}(l) = \sigma_{out}(l) \end{aligned}$$

The rest is standard direct-style denotational semantics.

The preconditions and postconditions generated by our algorithm are always in a canonical form:

$$\begin{aligned} \text{Precondition} \quad P &::= \bigwedge_i l_i = e_i \\ \text{Postcondition} \quad Q &::= \bigwedge_i l_i = \mathbf{in} \ e_i \wedge e = e' \wedge \mathit{modifies} \ \mathcal{L} \end{aligned}$$

In both cases, the predicate describes the values stored in some locations  $l_i$ . The postcondition describes the contents of locations  $l_i$  in the final state, in terms of expressions  $\mathbf{in} \ e$  evaluated in the initial state.<sup>8</sup> The postcondition also has a distinguished predicate  $e = e'$  that is used to assert the equality of two expressions before applying an algebraic law. In some cases, the distinguished predicate is unnecessary, in which case it can be replaced by an expression that is trivially true; for purposes of exposition, we elide trivial predicates. The interpretation of the *modifies* clause is the set of locations  $\mathcal{L}$  that are modified by the graph  $G$ .

If we assume that the graph and predicates have already been rewritten as necessary to satisfy the constraints, the resulting model is pleasantly simple: an initial state  $\sigma_{in}$  satisfying the precondition  $P$  must result in a final state satisfying the postcondition  $Q$  and modifying only locations in  $\mathcal{L}$ .

$$\begin{aligned} \emptyset \vdash \{P\} G \{Q \wedge \mathit{modifies} \ \mathcal{L}\} &\Leftrightarrow \\ \forall \sigma_{in} : \llbracket P \rrbracket (\sigma_{in}) \Rightarrow \exists \sigma_{out} : &(\llbracket G \rrbracket \sigma_{in} = \sigma_{out} \wedge \llbracket Q \rrbracket (\sigma_{in}, \sigma_{out}) \wedge \\ &\forall l \notin \mathcal{L} : \sigma_{in}(l) = \sigma_{out}(l)) \end{aligned}$$

Now that we have defined the judgment that must be computed by our algorithm, we can explain how the algorithm works.

<sup>7</sup>Hoare's work admits of nondeterministic programs in which the interpretation is a relation, not a function, with the slightly different type  $\llbracket G \rrbracket : \text{state} \times \text{state} \rightarrow \text{bool}$ . Our work carries through to this setting, but for the purpose of generating a back end, it is simpler to work only with deterministic graphs.

<sup>8</sup>We lift the  $\mathbf{in}$  operator to expressions using the natural mapping to apply  $\mathbf{in}$  to each location in the expression.

### 6.3.3 Algorithm overview

We initialize the pool (line 1 in Figure 6.2 on page 93) with RTLs from the  $\lambda$ -RTL machine description. We run most of the analyses in Chapter 4, with the exception of the compile-time constant analysis in Section 4.2.1 on page 44: this analysis is useful only for the recognizer, and it introduces constraints on the RTLs, which would unnecessarily complicate our algorithm for generating the tile set. For each RTL  $l_1 := e_1 \mid \cdots \mid l_n := e_n$  in the machine description, we can add the instruction to the pool with the following judgment:

$$\emptyset \vdash \{true\} \ l_1 := e_1 \mid \cdots \mid l_n := e_n \ \{l_1 = \mathbf{in} \ e_1 \wedge \cdots \wedge l_n = \mathbf{in} \ e_n \wedge \text{modifies} \ \{l_1, \dots, l_n\}\}$$

But because not all RTLs are useful for implementing tiles, we add only those that pass a utility test, which we define in Section 6.5.

The outermost loop (line 2) ensures that the algorithm continues until it can no longer add new implementations to the pool. Consequently, the details of ensuring termination are intimately related to the utility test that limits the RTLs added to the pool (Section 6.5).

Each iteration of the algorithm begins by using the implementations in the pool to construct a new data-movement graph (line 3). Because each iteration of the algorithm may discover new data-movement instructions, the graph from the previous iteration may be obsolete.

Our algorithm searches for new implementations using three nested loops (lines 4–6) that successively select an implemented RTL from the pool, a subexpression of the RTL, and an algebraic law to rewrite the subexpression.

The outer loop (line 4) is a linear iteration over the contents of the pool, selecting an implementation  $G_0$  of an RTL containing the effect  $l_0 := e_0$ . But because our algorithm continually adds implementations to the pool, one linear iteration is not enough: we have to iterate over the pool until our algorithm no longer finds new implementations.

The result is a breadth-first exploration of the capabilities of the machine: each iteration through the pool uses the implementations discovered in the previous iteration to find new implementations of RTLs. For example, the early iterations find implementations of the data-movement RTLs. These RTLs are used to construct a data-movement graph before each iteration begins. As enough data-movement RTLs are discovered, the data-movement graph becomes more complete, allowing us to find implementations of new RTLs by compensating for extra assignments in old RTLs. Then, we can use the resulting RTLs in the next iteration, and so on. Table 6.5 shows an

Initial pool:	
$r_1 := \text{sx}(\text{lobits}_{16}(k_{32}))$	load small signed immediate
$r_1 := r_2 \vee \text{zx}(\text{lobits}_{16}(k))$	bitwise-or immediate
$r_1 := r_2 \vee (k \gg_l 16) \ll 16$	load upper immediate
Round 1 finds load-immediate 0 and move	
$r_1 := r_2 \vee \text{zx}(\text{lobits}_{16}(0))$	implements $r_1 := r_2$
$r_1 := \text{sx}(\text{lobits}_{16}(0))$	implements $r_1 := 0$
Round 2 finds load-immediate $k$	
$t_1 := \text{sx}(\text{lobits}_{16}(0));$	
$t_2 := t_1 \vee (k \gg_l 16) \ll 16;$	
$r_3 := t_2 \vee \text{zx}(\text{lobits}_{16}(k))$	implements $r_3 := k$

Table 6.5: An example from the PowerPC showing how instructions found in one round can be used to implement new RTLs in subsequent rounds. The metavariables  $r_1$ ,  $r_2$ , and  $r_3$  stand for integer registers; the metavariables  $t_1$  and  $t_2$  stand for pseudoregisters. The initial pool contains the basic instructions used to implement a load-immediate RTL, but it is not yet clear how they can be combined. In round 1, our algorithm discovers that it can implement a move RTL and an RTL loading the immediate 0 in a register. Although we do not show its use in the next round, the move RTL is of particular note because it can be used to save and restore values in a register. In round 2, our algorithm finds an implementation of an RTL to load a 32-bit immediate, using implementations discovered in round 1.

example of how an implementation found in one round can be used to find new implementations in subsequent rounds. In the example, the metavariables  $r_1$ ,  $r_2$ , and  $r_3$  stand for integer registers, and the metavariables  $t_1$  and  $t_2$  stand for pseudoregisters. As we will see in Section 6.5, the number of rounds is bounded.

The next loop (line 5) is a straightforward depth-first search over the expression  $e_0$  computed by the implementation  $G_0$ . The depth-first search walks over the expression  $e_0$  computed by  $G_0$  and selects a subexpression  $e$  (possibly  $e_0$  itself).

The innermost loop (line 6) is a linear iteration over the set of algebraic laws; each iteration selects a single law  $e_1 = e_2$ . Of course, because the goal is to establish a state in which  $e = e_1$ , we do not attempt to apply every law to  $e$ . We always apply the trivial law  $x = x$ . If  $e$  is the application of an

operator  $\oplus$ , we also apply algebraic laws where the expression  $e_1$  is also the application of the same operator  $\oplus$ .

The next step (line 7) is the most complicated part of the algorithm: we try to find a set of constraints  $\mathcal{C}$  and a graph  $G$  that *establishes* a state in which  $e = e_1$ . After we have established a state in which  $e = e_1$ , we will be able to apply the algebraic law to conclude that  $e = e_2$ , and in turn, that  $E[e] = E[e_2]$ . Because the establishment step is a complex but important contribution of our algorithm, we explain it in depth in its own section (Section 6.3.4).

After establishment, we compensate for extra assignments (line 8), producing new graphs to save and restore locations. Most of the extra assignments come from the graph  $G$  that is used for establishment. But sometimes, we may want to compensate for an extra effect of  $G_0$ ; for example, if the instruction modifies the condition codes in addition to a desirable effect, we can compensate for the effect to the condition codes. In some cases, we may be able to compensate for an extra effect without saving and restoring the location: we may conclude that the extra effect modifies only an unobservable location. In fact, we can ensure that the location is unobservable by generating a constraint to ensure that a location is equal to a fresh temporary.

Having applied the algebraic law and compensated for extra assignments, we conclude that the implementation computes the expression  $E[e_2]$ , and *normalize* the judgment by eliminating the constraints (lines 9–10). The constraints are eliminated by substitution for the metavariables in the judgment.

But not every implementation discovered by our algorithm is added to the pool: if an implementation does not appear useful, we discard the implementation (line 11). We call this step *pruning*, and because the decision of whether an implementation is useful is intimately related to ensuring that the algorithm terminates, we discuss the topic separately in Section 6.5. If we decide to keep the implementation, then we add it to the pool (line 12), for use in the next iteration.

Now that we have explained the algorithm, we can focus on the most important step: establishment.

### 6.3.4 Establishment

The goal of the establishment procedure is to find a sequence of instructions that establish a state in which a predicate  $e = e_1$  is satisfied. In some ways, establishment is like unification: equalities may ensure that  $e$  and  $e_1$  are equivalent. But in establishment, the equalities in question involve the

contents of machine locations. These contents cannot be affected by binding values to constraint variables; they must be mutated by executing code.

We implement establishment using the two mutually recursive procedures in Figure 6.6 on page 100: the *establish* procedure (lines 1–23) attempts to unify two expressions  $e$  and  $e_0$ , and the *establishContents* procedure (lines 24–36) attempts to find a sequence of implementations from the pool that will store an expression  $e'$  in a location  $l$ . Each of the procedures in Figure 6.6 returns a graph used to establish the equality of the argument expressions. But sometimes such a graph cannot be found, in which case the pseudocode states that the procedure fails.

The programming model in the pseudocode is one of angelic nondeterminism: it is assumed that the procedures find every successful result and avoid any execution that fails.<sup>9</sup> For example, if one arm of a case statement fails, another applicable case may be executed. Constraints are also added with angelic nondeterminism. For example, in the process of establishing an equality, a new constraint may be added as a side effect, but we assume the constraint is added only if it will not make the set of constraints unsolvable. Similarly, we assume that any constraints added during failing executions will be discarded. And each time we use an algebraic law, an implementation from the pool, or an implementation from the data-movement graph, we assume that the metavariables are freshened. In practice, the implementation freshens metavariables by replacing them with newly generated unique names.

The *establish* procedure proceeds by structural induction on the expression  $e_0$ . Each of the cases on  $e_0$  tries the same two strategies: we can check whether  $e$  and  $e_0$  can be unified with a constraint, and if  $e$  is a location, we can use instructions from the pool to establish  $l = e_0$ . Because we may not be able to solve the constraints we generate, we treat the case statements in the algorithm as being non-deterministic: even if a constraint may be applied, our algorithm also tries to use instructions from the pool.

Although each case follows the same pattern, there are minor variations. In the case for constants (lines 3–8 on page 100), we first check whether we are trying to establish the equality of two identical constants, in which case no further work is necessary (line 4). In the case for locations (lines 9–13), we can only use a constraint if the location variable refers to a location set containing the other location  $l$  (line 11).

---

<sup>9</sup>The implementation finds every successful result by executing every iteration of the `foreach` loops, instead of stopping when a `return` statement is reached. Consequently, the implementation requires extra bookkeeping to keep track of the accumulated results from every iteration of a loop.

```

1 establish( $e, e_0$ ) :  $e \times e \rightarrow G$ , such that  $G$  establishes  $e = e_0$ 
2 case  $e_0$  of:
3    $k$ :
4     if  $e = k$  then return  $\epsilon$ 
5     else case  $e$  of:
6        $x_k$ : add constraint  $x_x = k$ ; return  $\epsilon$ 
7        $l'$ : return establishContents( $l', k$ )
8       default: fail
9    $l$ :
10    case  $e$  of:
11       $x_l:ls$ : if  $l \in ls$  then (add constraint  $x_l:ls = l$ ; return  $\epsilon$ ) else fail
12       $l'$ : return establishContents( $l', l$ )
13      default: fail
14   $\oplus(e_1, \dots, e_n)$  :
15    case  $e$  of:
16       $x_k$ : if compileTime( $e$ ) then (add constraint  $x_k = e$ ; return  $\epsilon$ ) else fail
17       $\oplus(e'_1, \dots, e'_n)$  :
18        Let  $G_1, \dots, G_n = \textit{establish}(e'_1, e_1), \dots, \textit{establish}(e'_n, e_n)$ 
19        if  $\forall i$  : modifies( $G_i$ )  $\parallel$  uses( $e_1$ )  $\cup \dots \cup$  uses( $e_{i-1}$ ) then return  $G_1; \dots; G_n$ 
20        else fail
21       $l'$ : return establishContents( $l', \oplus(e_1, \dots, e_n)$ )
22      default: fail
23     $x_e$ : add constraint  $x_e = e$ ; return  $\epsilon$ 
24 establishContents( $l, e'$ ) :  $l \times e \rightarrow G$ , such that  $G$  establishes  $l = \textit{in } e'$ 
25 case  $e'$  of:
26    $l'$ : if ( $l' \rightarrow l, G_{\textit{move}}$ )  $\in$  DataMovementGraph then return  $G_{\textit{move}}$  else fail
27    $k$ :
28     foreach implementation  $\vdash \{true\} G \{l' = k' \wedge \textit{modifies } \mathcal{L}\}$  in the pool:
29       Let  $G' = \textit{establish}(k', k)$  // Note:  $k$  or  $k'$  may be a constraint variable
30       if ( $l' \rightarrow l, G_{\textit{move}}$ )  $\in$  DataMovementGraph then return  $G'; G; G_{\textit{move}}$ 
31       else fail
32    $\oplus(e_1, \dots, e_n)$ :
33     foreach implementation  $\vdash \{true\} G \{l' = \oplus(e'_1, \dots, e'_n) \wedge \textit{modifies } \mathcal{L}\}$  in pool:
34       Let  $G' = \textit{establish}(\oplus(e'_1, \dots, e'_n), \oplus(e_1, \dots, e_n))$ 
35       if ( $l' \rightarrow l, G_{\textit{move}}$ )  $\in$  DataMovementGraph then return  $G'; G; G_{\textit{move}}$ 
36       else fail

```

Figure 6.6: The establishment procedure returns a graph that establishes a state in which  $e = e_0$ . The helper function *establishContents* finds implementations from the pool to store the expression  $e'$  in  $l$ . In both functions, case statements are angelically non-deterministic: if one case fails, another applicable case may be applied. Similarly, adding a constraint fails if it would make the set of constraints unsolvable. Each time an implementation is drawn from the pool (lines 28, 33) or from the data-movement graph (lines 26, 30, 35), we freshen all the metavariables.

The case for operator application is more complex because it is inductive (lines 14–22). If the operator application is a compile-time expression and the expression  $e$  is a compile-time constant, we may be able to use a constraint to establish  $e = e_0$ . Otherwise, if  $e$  is an application of the same operator  $\oplus$ , we call the *establish* procedure recursively to establish  $e'_i = e_i$  for each of the subexpressions of  $e$  and  $e_0$ . We can then sequence the resulting graphs  $G_1; \dots; G_n$ , as long as no graph overwrites a live location defined by a previous graph. We conservatively estimate the live locations as the set of locations that will be used in the expression  $e_i$ .

If we are required to establish  $l = e'$ , then we use the *establishContents* procedure to look for implementations in the pool. The easy case is when  $e'$  is a location  $l'$ , in which case we look for a suitable move instruction in the data-movement graph. Otherwise, we iterate over the pool, looking for each plausible implementation  $G$ : if  $e'$  is a constant  $k$ , then we consider implementations that compute constants (line 28); if  $e'$  is an application of the operator  $\oplus$ , then we consider implementations that also compute applications of the operator  $\oplus$  (line 33). Then, we call the *establish* procedure to find a graph  $G'$  such that the sequence  $G'; G$  computes the expression  $e'$ . (lines 29 and 34). Next, because  $G$  stores its result in a location  $l'$ , we find a move instruction  $G_{move}$  to move the result to  $l$ . The resulting sequence  $G'; G; G_{move}$  stores the expression  $e'$  in the location  $l$  (lines 30 and 35).

We guarantee the termination of our establishment algorithm by ensuring either that *establish* is called with arguments that return without any further recursive calls (line 29), or that each recursive call to *establish* passes a smaller expression as the argument  $e_0$  (line 34).

## 6.4 Discussion

Our algorithm relies on two important invariants:

- Each implementation in the pool makes explicit claims about the assignments it performs, ensuring that any other assignments are unobservable.
- Every RTL produced by our algorithm can be implemented by a sequence of instructions on the target machine.

These invariants are exactly what the tiler requires, and they are also essential for the establishment procedure to correctly compose a sequence of machine instructions that compute a desirable expression.

But correctness is not enough: our algorithm must not be too slow. Because our algorithm (Figure 6.2 on page 93) is built around three nested

loops (lines 4–6), it might appear that the search space is prohibitively large. But in practice, the search space is limited: with our current set of algebraic laws, at most 5 laws may apply to any particular expression  $e$ , and because every implementation produced by our algorithm contains only machine instructions, the search space is also limited by the set of instructions on the target machine. The search space is further cut down by the pruning procedure in Section 6.5. As a result of these restrictions, our algorithm generates a tileset in just a few minutes (Section 6.6 on page 106).

## 6.5 Pruning and termination

We have explained how to find implementations of new RTLs, but how do we know when to stop? Unfortunately, the underlying problem is undecidable: there is no terminating algorithm that is guaranteed to find an implementation of a tile if one exists (see Appendix A). The intuition behind the result is that the problem of finding an implementation that is equivalent to a tile is closely related to the more general, undecidable problem of determining if two programs are equivalent. Our proof proceeds by reduction from the halting problem; we adapted well-known proofs of undecidability for strong normalization in term-rewriting systems (Huet and Lankford 1978; Klop 1992; Bezem, Klop, and Roel de Vrijer 2003).

Given the undecidability, our goals are to limit the RTLs added to the pool in such a way that:

- On real machines, we are likely to find implementations of all the tiles.
- The total number of implementations added to the pool is small enough that the tileset generator runs in minutes, not hours.
- The implementations are sequences of instructions that expose opportunities to the optimizer.

As we discuss in Section 7.2 on page 111, these criteria are not satisfied by standard heuristics that limit either the size of an implemented RTL or the depth of the search conducted by our algorithm. Instead, we have developed a novel heuristic that estimates how likely it is that an RTL will contribute to an implementation of a tile.

Given an RTL, our heuristic predicts *the number of algebraic laws that may have to be applied* to produce an implementation of a tile. Our heuristic is founded in the observation that real machines are designed with compilers in mind: to compute a simple instruction on a real machine, you might have

```

1 estimateNumLaws( $\emptyset \vdash \{true\} G \{\bigwedge_{i \leq n} l_i = \mathbf{in} e_i \wedge \text{modifies } \mathcal{L}\}$ ) : impl  $\rightarrow$  int
2   return  $\min\{expEstimate(e_i) \mid i \in \overline{1..n}\}$ 
3
4 expEstimate(e) : e  $\rightarrow$  int
5   if e is the expression computed by an expansion tile then return 0
6   else return  $\min\{1 + coverWithAlgLawFragment(e, e') \mid e_x = e_y \in \text{Laws}, e' \succ e_x\}$ 
7
8 coverWithAlgLawFragment(e, e') : e  $\times$  e  $\rightarrow$  int
9   case (e, e') of:
10     ( $\oplus(e_1, \dots, e_n), \oplus(e'_1, \dots, e'_n)$ ):
11       return  $\sum_{i=1}^n \min(coverWithAlgLawFragment(e_i, e'_i), expEstimate(e_i))$ 
12     default: return  $\infty$ 

```

Figure 6.7: Pseudocode describing our heuristic for estimating the number of algebraic laws that must be applied before our algorithm may conclude that the input RTL is used in an implementation of a tile.

to put together a long sequence of machine instructions, but you probably won't have to do much reasoning beyond forward substitution. Many interesting low-level computations can be implemented by sequences of such very simple instructions as shift operations, logical operations, and arithmetic operations (Warren 2002). So instead of limiting the length of such sequences or the complexity of the expressions they compute, we limit the amount of *reasoning* that required to show that a sequence of instructions is useful. We postulate that the more algebraic laws have to be applied, the less likely an implementation is to be used to implement a tile.

Because we cannot know ahead of time the actual number of algebraic laws we may usefully apply to an expression, our heuristic estimates by trying to cover the expression with fragments of algebraic laws. The idea of covering the expression with algebraic laws is based on the observation that our algorithm finds implementations of expansion tiles by applying algebraic laws to rewrite implementations in the pool. Therefore, a candidate should only be added to the pool if it can be rewritten using algebraic laws to implement an expansion tile. But because the establishment procedure may sequence multiple instructions to establish a state in which an algebraic law may be applied, a candidate need only compute a fragment of the expression that is rewritten by the algebraic law. For example, we can use the boldfaced and underlined fragments of the following algebraic laws<sup>10</sup>

<sup>10</sup>The  $gx_w$  (“garbage-extend”) operator widens a value to a width  $w$  with the low bits containing the argument and the high bits containing unspecified values. It is not safe to make any assumptions about the high bits in the resulting value. This operator is use-

$\text{x86\_carrybit}(\text{x86\_setcarry}(x_e)) = x_e$   
 $\text{x86\_flags2ah}(\text{x86\_sbbflags}(x_e, x'_e, x''_e)) = \text{gx}(\text{borrow}(x_e, x'_e, x''_e))$   
 to cover the boldface and underlined parts of the following expression  
 $\text{x86\_sbbflags}(r_1, r_2, \text{x86\_carrybit}(\text{x86\_setcarry}(r_3 @ 0:1 \text{ bits})))$

The remaining uncovered expressions are expressions computed by expansion tiles: in our case, the uncovered expressions are fetches from locations, which are computed by data-movement tiles. Therefore, we conclude that the expression can be covered using two algebraic laws. The covering that uses the fewest number of algebraic laws is taken as a conservative estimate for how likely it is that a candidate may help implement an expansion tile.

Our heuristic is described by the high-level pseudocode in Figure 6.7: the function *estimateNumLaws* takes a candidate for the pool and returns an estimate of the number of algebraic laws that must be applied before the candidate may yield an expansion tile. We compute an estimate for each assignment made by the candidate (line 2 in Figure 6.7). Because our algorithm may eventually compensate for the other assignments, a single promising assignment is enough to keep the candidate in the pool.

To estimate the number of algebraic laws required to cover an expression  $e$ , we use the mutually recursive functions *expEstimate* (lines 4–6) and *coverWithAlgLawFragment* (lines 8–12). If  $e$  is an expression computed by an expansion tile (line 5), then no algebraic laws are required. Otherwise, we try to find the least number of algebraic laws that can cover the expression  $e$  (line 6). But because the application of an algebraic law  $e_x = e_y$  may require the establishment procedure to find multiple instructions to compute the expression  $e_x$ , we have to try covering  $e$  not just with  $e_x$  but also with each subexpression  $e'$  of  $e_x$ . We count the algebraic law and call *coverWithAlgLawFragment* to try covering  $e$  with  $e'$ . We ignore the expression  $e_y$  because our heuristic cares only that the expression can be rewritten, not about the result of the rewrite.

The function *coverWithAlgLawFragment* attempts to cover the expression  $e$  with an expression  $e'$  that is a fragment of an algebraic law. If both expressions are applications of the same operator, then we have to try to cover each of the subexpressions of  $e$ . We may be able to cover a subexpression  $e_i$  with a subexpression  $e'_i$  of the algebraic law. But we can also try to cover  $e_i$  with another algebraic law by calling *expEstimate* recursively. Either one of these options may produce a covering of  $e_i$ , so we try both and choose the cheaper option for each subexpression (line 11). The recursion is guaranteed to terminate because every recursive call to *expEstimate* reduces the size of the input expression  $e$ .

---

ful for describing an expression  $\hat{e}$  (Figure 5.1 on page 58) where only the low bits are important.

We avoid having a special case to check whether  $e$  is a constant or a fetch from a location by requiring that the caller handle those cases:

- The first call (line 6) occurs in *expEstimate*, which has already checked whether  $e$  is computed by an expansion tile, including a constant or location (line 5).
- The other call (line 11) occurs in an expression that computes two estimates and keeps the minimum. Because the other estimate by *expEstimate* will be 0 if the expression is a constant or fetch, the estimate returned by *coverWithAlgLawFragment* is irrelevant.

In practice, we have found that a maximum covering of size 4 is sufficient to implement the tileset on the ARM, PowerPC, and x86. Therefore, we add a candidate to the pool only if our heuristic estimates that we might find the implementation of an expansion tile by applying at most 4 algebraic laws. If our heuristic estimates that more than 4 laws will be required, we discard the candidate.

By discarding overly complicated candidates, we ensure that the size of the search space is bounded and the algorithm terminates. Every implementation added to the pool computes an expression  $e_0$  (line 4, Figure 6.2 on page 93) that has passed the utility test. And to pass the utility test, we must have covered  $e_0$  with a limited number of fragments of algebraic laws. Because each algebraic law is of finite size, there is an upper bound on the size of the largest expression that can be tiled with a limited number of fragments. Furthermore, if the size of the largest expression must be finite, then the number of distinct expressions of that size must also be finite, which means that there is an upper bound on the number of distinct expressions  $e_0$  that can be added to the pool.

Although the pruning heuristic bounds the number of distinct expressions we implement, we still have to ensure that the algorithm does not add the same expression to the pool repeatedly. We could check whether the pool already contains an implementation of an expression by a straightforward search for the expression. But we can do even better: sometimes an implementation of an expression such as  $r_I + k$  can be used in any context where another expression such as  $r_I + 1$  can be used. In those cases, we say that the first expression *subsumes* the first. Usually, we would like to keep only the first expression in the pool.

To identify opportunities for subsumption, each time we add an implementation of an expression  $e$  to the pool, we gather all the expressions in the pool that compute the same expression as  $e$ , ignoring locations and constants, which may differ. We then check whether  $e$  subsumes any of the

other expressions or if  $e$  is subsumed by any of the other expressions. The pairwise check for subsumption reuses our implementation of establishment, but we run it with a pool containing only implementations of data-movement and load-immediate instructions. If the implementation of  $e$  can establish  $e'$  without using other instructions from the pool, then the implementation of  $e$  subsumes  $e'$ .

But subsumption is not all that matters; sometimes we want to keep both implementations because the subsumed implementation is better in some cases. We use a simple cost function to exclude an implementation from the pool only if it is more expensive than other possible implementations. Our cost function predicts execution time, giving a low cost to each register access and a high cost to each memory access.<sup>11</sup> But because different implementations may modify different sets of scratch registers, a more expensive implementation may be useful to find an implementations when we cannot use a cheaper implementation that modifies a different set of scratch registers. Therefore, we exclude an implementation  $i$  from the pool only if there is a cheaper, subsuming implementation  $i'$  that modifies a subset of the scratch registers modified by  $i$ . Because there can only be a finite number of distinct sets of scratch registers, the number of implementations that we keep for each expression in the pool must be finite, so the algorithm must terminate.

## 6.6 Pragmatics of tileset generation

If we only ran the tileset generator once for each target machine, the speed of the tileset generator would not matter. But because the process of developing a new back end is iterative, the tileset generator must be fast.

The typical process of developing a back end is to implement the tileset incrementally. We write a portion of the machine description, generate the instruction selector, and run the compiler's test suite. When we run the test suite, the compiler complains each time it tries to use a tile that has not been implemented. We then add the relevant instructions to the machine description,<sup>12</sup> write any necessary algebraic laws, and generate the instruction selector anew.

---

<sup>11</sup>Davidson has suggested that the cost function should be based not on execution costs but on the simplicity of the implementation, because a simpler implementation provides more opportunities for the optimizer to improve the entire procedure (Davidson 2008a). Fortunately, the cost function is trivially replaced, allowing for future experimentation.

<sup>12</sup>The lazy compiler writer can omit the binary encoding of the instruction, since it is not necessary for our purposes.

Target	Search Time (s)	Total Time (s)
ARM	3.09	53.32
PowerPC	36.72	60.50
x86	336.16	363.11

Table 6.8: Time required to generate an instruction selector: For each generated back end, we list the time spent in the heuristic search, as well as the total time spent generating an instruction selector. The back ends were generated on a relatively modest Intel Pentium M, 1.5 GHz with 1 GB of memory.

Because the tileset generator is part of the development cycle, it must be reasonably fast. In Table 6.8 on page 107, we list the time required to generate an instruction selector, along with the time spent in the heuristic search. Each of the instruction selectors is generated within minutes, with the *x86* instruction selector requiring the most time at about 6 minutes on a relatively modest 1.5 GHz Pentium M.

We can see the incremental progress of our search algorithm in Table 6.9 on page 108, which lists the number of implementations produced at each outer iteration of the algorithm (i.e. line 2 of Figure 6.2 on page 93). Due to the size of the *x86* instruction set, the pool grows largest when generating an instruction selector for the *x86*. It is worth noting that the number of iterations is greater than the maximum cost permitted by our pruning heuristic. The reason is that each time we find an implementation of one tile, it might lead the search closer to the implementation of a different tile. For example, on some machines, we can only find an implementation of the remainder operator after we have found an implementation of the division operator. It is also worth noting that on both the ARM and the PowerPC, the number of implementations after the first iteration is smaller than the number of implementations after initialization. The reduction in implementations is due to subsumption: some of the implementations found in the first iteration subsume implementations in the initial pool.

The progress of the search is influenced greatly by the set of algebraic laws. The algebraic laws affects not only how the search finds new implementations, but also whether the pruning heuristic accepts each new implementation. Because of these dual effects, it is difficult to characterize how each additional algebraic law affects the time required to generate an instruction selector. But the most frequent reason to add a new algebraic laws is to make use of a machine-dependent operator, and our algorithm's establishment procedure can only make use of a machine-dependent law on

Iteration	ARM	PowerPC	x86
Initial	426	275	863
1	368	213	1284
2	474	280	1412
3	491	315	1639
4	491	387	1703
5		421	1713
6		454	1727
7		462	1733
8		466	1733
9		466	

Table 6.9: Expansion of the pool by iterations: For each generated back end, we list the number of implementations in the pool at initialization and after each iteration (line 2 of Figure 6.2 on page 93).

the machine with the machine-dependent operator. Therefore, a machine-dependent algebraic law only affects the time required to generate an instruction selector for a single target machine.

As for algebraic laws that are not machine-specific, our experience has been that an algebraic law has a greater impact on the efficiency of the search when the algebraic law leads to the establishment procedure drawing more than one instruction from the pool. For example, a costly algebraic law might show that a sequence of shifts can implement a sign-extension from  $n$  bits to  $w$  bits:

$$(x \ll (w - n)) \gg_a (w - n) = \text{sx}(\text{lobits}(x))$$

This law is costly because the establishment procedure may have to combine implementations from the pool for of up to four expressions: a shift left, a shift right, and two subtractions. There may be multiple implementations for each of these expressions, and the establishment procedure will try every combination. Of course, the time spent applying these algebraic laws is mitigated substantially by our use of subsumption to avoid keeping redundant implementations in the pool.

## Chapter 7

# Search and termination in tileset generation

Many others have attempted to automate the generation of an instruction selector, but instead of trying to solve the undecidable problem of searching for implementations, they rely on the compiler writer to solve the hard problem of mapping the compiler's IR onto machine instructions. A notable exception is Cattell (1982). The only real approach to solving an undecidable problem is heuristic search, but Cattell's search strategy is quite different from ours.

The terminology in Cattell's dissertation is specific to his compiler, but for the sake of exposition, I describe his algorithm using the terms defined in this dissertation. Specifically, I use the term *instruction* only to refer to an instruction on the target machine. To refer to a computation that may or may not be representable by a single instruction on the target machine, I use the term *RTL*.

Cattell's algorithm conducts a search in the opposite direction from our algorithm: his search begins with an expansion tile, which is rewritten at each step to an equivalent sequence of RTLs, until it has been rewritten to a sequence of machine instructions. Each semantics-preserving transformation applies one of several laws. Most of the laws are similar to our algebraic laws, and some of the *decomposition* laws are used to reason about how a sequence of RTLs can be equivalent to a single RTL. The key to Cattell's algorithm is to choose transformations that will eventually lead to a sequence of equivalent machine instructions.

The choice of transformations is made using a means-ends search. First, a heuristic chooses a set of machine instructions that might plausibly be used to implement the tile. Cattell's heuristic decides whether an instruction is plausible by considering the root of the expression it computes. For exam-

ple, if an instruction computes the expression  $r_2 + (r_3 * r_4)$ , then the root is the operator  $+$ ; if an instruction is a constant  $k$ , then the root is  $k$ . An instruction is considered plausible either if it has the same root as the tile or if there exists an algebraic law  $e_1 = e_2$  where both  $e_1$  and  $e_2$  have the same root operators as the tile and the machine instruction. Because the entire, unbounded search space cannot be explored, the heuristic also orders the plausible instructions based on an estimate of similarity between each plausible instruction and the tile.

After a set of plausible instructions is selected, Cattell's algorithm attempts to transform the tile to a sequence of machine instructions. The algorithm proceeds by induction on the tile and the plausible instruction, attempting at each step to ensure that the roots of the expressions are the same. If they are not the same, then the algorithm rewrites the the expression in the tile using an algebraic law such that the roots of the expressions will be the same. In some cases, the tile cannot be rewritten to a machine instruction, in which case the tile may be decomposed into a sequence of two RTLs; the search algorithm begins anew, trying to find machine instructions for each RTL in the sequence.

Like our algorithm, the search space for Cattell's algorithm is unbounded, so the search must be pruned. Cattell uses two tunable criteria: the breadth of the search and the depth of the search. The breadth of the search limits the number of plausible instructions that are produced by his heuristic. The depth of the search limits the number of times his algorithm may apply an algebraic law or decompose the RTL into multiple RTLs.

Although our algorithm and Cattell's algorithm share basic ideas like using algebraic laws and searching for a sequence of machine instructions, there are two differences that merit further discussion:

- *Search strategy:* Cattell's algorithm starts with expansion tiles and searches for an implementation using machine instructions; our algorithm starts with machine instructions and searches for implementations of new RTLs, eventually producing expansion tiles. Our search strategy allows a simpler implementation by avoiding the need for a heuristic to guide the search toward an implementation using machine instructions.
- *Pruning:* Cattell's algorithm prunes the search by limiting the breadth and depth of the search; our algorithm prunes the search by discarding implementations that don't appear useful. Our pruning strategy allows long sequences of instructions that expose opportunities to the optimizer.

In the rest of this chapter, we discuss the tradeoffs involved in the search strategies and pruning strategies used in both Cattell's algorithm and our algorithm.

## 7.1 Search Strategies

Because our algorithm searches for tiles in the opposite direction from Cattell's algorithm, we are able to avoid much of the complexity required in Cattell's search procedure. Cattell's search begins with a tile, then the search rewrites the tile until an equivalent sequence of machine instructions is found. Without the heuristic to guide the search toward machine instructions, the search space would be prohibitively large because it would include all semantically equivalent sequences of RTLs. Our algorithm does not need a heuristic to guide the search toward machine instructions because our search begins with machine instructions, and we maintain the invariant that every implementation in the pool consists of machine instructions. But why doesn't our algorithm need a heuristic to guide our search toward tiles?

By design, the goal of our search is already very close to the starting point. The tiles are designed to express basic machine instructions on as many targets as possible. Therefore, a relatively shallow search is sufficient to find a sequence of machine instructions to implement a tile. And because we only consider sequences of instructions that are valid on the target machine, our search space is constrained to RTLs that can be implemented on the target machine.

The benefit of our approach is that we do not need to implement any heuristics to guide our search towards a tile. Instead, our algorithm can use a straightforward search strategy that applies each algebraic law at every opportunity. Pruning the search is sufficient to prevent the search from wandering too far away from the tiles.

## 7.2 Pruning Strategies

There are two standard approaches to pruning a search: dynamic heuristics and static heuristics. Because we were not satisfied with tradeoffs of either technique, we developed a hybrid heuristic with the benefits of both approaches.

A dynamic heuristic directly constrains how the algorithm explores the search space. For example, Cattell's algorithm limits the depth and breadth of the search space. Another popular type of dynamic heuristic is to require each step of the algorithm to reduce the size of the problem; for example, we

could restrict each step of our algorithm to produce an implementation of a smaller RTL. But this restriction may prevent our algorithm from finding implementations: it is often necessary to construct a sequence of instructions that computes a larger RTL before we can apply a useful algebraic law.

The benefit of a dynamic heuristic is that it can directly bound the amount of effort expended by the search: Cattell's algorithm limits the number of times it attempts to apply algebraic laws. But this restriction is very coarse: even though the application of an algebraic law may produce a useless result, the search still includes any results reachable within the depth and breadth limits.

A static heuristic, on the other hand, attempts to evaluate whether a state in the search space appears useful; states that do not seem useful can be pruned. Such a heuristic is called *static* because it evaluates a state independent of the search required to reach that state. For example, one possible heuristic would limit the height of the expression computed by an implementation. But in the presence of algebraic laws, the height of a tree is not a very good measure of whether it will help implement a tile. On some machines, we *need* high abstract syntax trees; for example, in order to load a 32-bit constant into a register on a typical RISC machine, we need to apply the algebraic law  $((e \gg_l n) \ll n) \vee \text{zx}(\text{lobits}_n(e)) = e$ , which is applicable only to an abstract-syntax tree of height at least 4.

Another possible heuristic would limit the number of machine instructions in an implementation. Limiting the number of instructions is explicitly at odds with our strategy of producing simple code in the expander, then relying on the optimizer to improve the code. If we limited the number of machine instructions that could be combined, our algorithm would have to find a short, efficient sequence of instructions, taking advantage of complex instructions and complex addressing modes. Not only is it more difficult to generate a tileset that uses the most efficient implementations, but it does not even result in better code. Such implementations, while ideal for a BURS tiling, are the worst possible input for a Davidson/Fraser optimizer, which does best when fed long sequences of very simple instructions (Davidson 2008b). And without the help of the optimizer, we cannot expect our approach to generate better code than previous work on generating BURG-style instruction selectors, in which generated instruction selectors have produced significantly slower code than hand-written instruction selectors (Chapter 9).

The benefit of a static heuristic is that it can evaluate the likelihood that the search algorithm has reached a promising branch of the search tree. But without considering the steps that must be taken by the search algorithm,

a static heuristic cannot predict whether the search algorithm will actually reach its goal.

Therefore, we designed a hybrid heuristic to prune our search. Like a static heuristic, our heuristic evaluates the implementation produced by each step of the algorithm. But the evaluation metric is designed as a quick estimate of a dynamic property: how many algebraic laws may be applied before the implementation will yield a tile. An important benefit of this hybrid heuristic is that we can apply it to the pool upon initialization of our algorithm, which means that we can cut down the search space before we have even begun to search.

We believe that this heuristic combines the benefits of both approaches: it evaluates the likelihood that an implementation may lead to a successful search, allowing us to prune the search when the effort to find a tile looks prohibitive.



## Chapter 8

# Costs and benefits of generating back ends

The ideas of lasting value in this dissertation are the algorithms and technical results presented in chapters 4-7, which show how to generate a back end automatically. But we need to know not only whether it is possible to generate a back end automatically but also whether this is a good way to retarget a compiler. We evaluate our process of retargeting a compiler by answering two questions:

1. Are the generated components suitable for use in a production compiler?
2. Is our process of retargeting a compiler easier than the processes used in well-known retargetable compilers?

One way to answer the first question is to insert the generated components in a production compiler and perform experiments. But using a production compiler is not as simple as plugging in the generated components: we would need to implement the tiler and mitigate the differences between our representation of RTLs and the production compiler's intermediate representation, not to mention the complications involved in working with a heavily engineered software artifact.

Instead, we have inserted the generated components into a research compiler. Unlike a production compiler, our research compiler has received many fewer man-hours of development; as a result, it cannot be expected to compile all the same programs that are compiled by a production compiler, nor can it be expected to produce the same quality of optimized code. But our compiler does compile many of the programs handled by a production compiler, producing code that is comparable to a nonoptimizing compiler.

Furthermore, the limiting factor in the number of programs our compiler can compile is not the back end of the compiler; it is the front end. For example, our C front end accepts only ANSI C programs, and our compiler cannot compile variadic functions; both of these limitations restrict our ability to compile standard benchmarks. Although it would be preferable to evaluate the generated back ends on a complete suite of standard benchmarks, constructing a complete front end for the compiler is outside the scope of this dissertation. Instead, we evaluate the compiler on a standard, but limited, set of benchmarks.

While it is relatively easy to assess whether the generated back ends work, it is more difficult to assess whether we have developed a good approach to retargeting a compiler. The ease of retargeting a compiler is a property that arises from many decisions that determine the structure of the compiler. To help answer this question, I identify a useful set of criteria, and I present a thin slice of the code generators in three well-known retargetable compilers, as well as both the hand-written and automatically generated back ends in our compiler. This assessment amounts to a case study comparing our method of retargeting a compiler with well-known techniques, and I leave conclusions about the best approach to the reader.

## 8.1 Quality of generated code

To show that the generated components of the back end are suitable for a production compiler, I have generated back ends for the *x86*, *PowerPC*, and *ARM* architectures. The *x86* and *ARM* are the dominant architectures among desktop and embedded computers, and the *PowerPC* is a representative RISC machine that was in prominent use until Apple stopped manufacturing computers with *PowerPC* processors in 2006.

To compile benchmarks written in C, we use a back end for the `gcc` compiler (Fraser and Hanson 1995) that emits C`--` (Peyton Jones, Ramsey, and Reig 1999; Ramsey and Peyton Jones 2000). The C`--` code is then compiled to assembly code by our compiler, `Quick C--`. Because `gcc` compiles only programs written in ANSI C, we can use benchmarks only if they are written in ANSI C, and because our C`--` does not admit variadic functions, we cannot compile benchmarks that require the definition of variadic functions.<sup>1</sup> Accordingly, we have chosen benchmarks from the SPEC CPU95 and SPEC CPU2006 benchmark suites that can be compiled by our `gcc` and `Quick C--` front ends. For each benchmark, we measure the wall-clock run-

---

<sup>1</sup>Although a program in C`--` cannot define a variadic function, it can call a variadic function such as `printf()`.

Benchmark	lcc	Hand / lcc	Generated / lcc	gcc -O0 / lcc	gcc -O1 / lcc
compress-95	43.32 s	0.84	0.80	1.01	0.69
go-95	25.84 s	0.98	0.88	1.01	0.61
vortex-95	55.84 s	1.09	1.10	1.05	0.77
mcf-2006	230.24 s	0.94	0.92	0.97	0.90
bzip2-2006	316.06 s	0.91	0.90	1.06	0.68

Table 8.1: Running times for benchmarks on *x86*: We use `lcc` as the baseline and give run times in seconds; other compilers are normalized to `lcc`. The Hand and Generated columns represent Quick C-- with hand-written and automatically generated *x86* back ends. Benchmarks were compiled without debugging or profiling information, then run on an AMD Athlon MP 2800+ with 2 GB of memory.

ning time, averaged over five runs. The CPU95 benchmarks show results on the reference set of inputs, and the CPU2006 benchmarks show results on the training set of inputs.

The experiments measure the run times of benchmarks compiled with the Quick C-- compiler using automatically generated back ends for the *x86* and PowerPC back ends. I have not run these benchmarks on the ARM because Quick C-- currently cannot compile programs to architectures that rely on software libraries to implement RTL operators that are not supported by the hardware (e.g. division, floating-point add). Although it would be possible to compile these programs by writing rewrite rules to help our search algorithm find implementations of these operators, that approach is not very practical, in part because it would lead to bloated executables. The standard technique is to implement the operators by making function calls to the software libraries that are provided by the manufacturers of the ARM. Until this support is added to Quick C--, we cannot run serious benchmarks on the ARM. Instead, I have verified that the ARM back end correctly compiles the integer-only benchmarks in the Quick C-- and `lcc` test suites.

To verify that the automatically generated *x86* back end produces code that is as good as a hand-written back end, I also compiled the benchmarks with Quick C-- using a hand-written back end for the *x86*. And to confirm that the generated back end produces code that is comparable to a production compiler with optimizations turned off, I performed the same experiments using `lcc` and `gcc`.

The experimental results in Table 8.1 verify that programs compiled by our automatically generated back end for the *x86* run at least as fast as the programs compiled by the hand-written back end, The one exception is the

<b>Benchmark</b>	<b>gcc -O0</b>	<b>gcc -O1 / gcc -O0</b>	<b>Generated / gcc -O0</b>
compress-95	107.07 s	0.39	0.72
go-95	62.03 s	0.40	0.73
vortex-95	144.81 s	0.54	0.73
mcf-2006	238.63 s	0.77	0.81
bzip2-2006	670.14 s	0.43	0.60

Table 8.2: Running times for benchmarks on PowerPC: We use `gcc -O0` as the baseline and give run times in seconds; other compilers are normalized to `gcc -O0`. The `Generated` column represents Quick C-- with an automatically generated PowerPC back end. Benchmarks were compiled without debugging or profiling information, then run on a PowerPC 7447A clocked at 1.25 GHz with 1 GB of memory.

vortex-95 benchmark, and the unnormalized results show that the difference in vortex-95 is less than three-tenths of one percent.

Similarly, the experiments show that most programs compiled by `lcc` and `gcc` without optimizations (`gcc -O0`) are faster when compiled by our generated back end; the one exception is the benchmark `vortex-95`. Although none of the compilers are performing many code-improving transformations, we hypothesize that the peephole optimizer in Quick C-- accounts for the superior speed of programs compiled with our generated back end. When `gcc` uses a variety of scalar optimizations (`gcc -O1`), it produces code that is 1.7% to 29.5% faster than code produced by our back end. Until we have implemented a set of optimizations in Quick C--, we cannot expect it to compete with an optimizing compiler such as `gcc`. But we can rely on previous work showing that the standard scalar and loop optimizations can be implemented in a Davidson/Fraser compiler like Quick C--, resulting in highly optimized code (Benitez and Davidson 1994).

I ran the same set of experiments on the PowerPC to evaluate Quick C-- with a generated back end for the PowerPC, in comparison with `gcc` with optimization levels `-O0` and `-O1`. The results of these experiments (Table 8.2) were similar to the experiments on the `x86`: programs compiled by the generated back end in Quick C-- were faster than those compiled by `gcc -O0` but slower than `gcc -O1`.

Of course, it is important not only that the compiled code is fast but also that the compiler itself is reasonably fast. In Section 4.3.3 on page 50, I showed that a generated recognizer is no slower than a hand-written recognizer. To test the speed of a generated instruction selector, I measured

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

Table 8.3: Compile times for generated and hand-written back ends for the *x86*: The experiments show the time required to compile the Quick C-- test suite (10,718 lines in 91 files), broken down by the time spent in the expander and total time. Times are averaged over five runs, with each run compiling the entire test suite to assembly code with one invocation of the compiler. Both versions of the compiler use a generated recognizer. The compiler ran on a relatively modest Intel Pentium M, 1.5 GHz with 1 GB of memory.

the time required to compile the Quick C-- test suite (10,718 lines of C-- in 91 files) using generated and hand-written instruction selectors. To reduce start-up overhead, the compiler was invoked once to compile the entire test suite to assembly code; the experiment was repeated five times, and the results show the average compile times (Table 8.3 on page 119).

The compile time for the generated instruction selector is only about 10% longer than the compile time for the hand-written instruction selector. The extra time spent in the generated expander is minimal; instead, most of the extra time is spent improving the more naïve code produced by the generated instruction selector.<sup>2</sup> In fact, the optimizer in Quick C-- has not undergone any performance tuning, so it may be possible that the small difference in compile times could be further reduced by improving the optimizer.

Having established that we can generate a back end that is suitable for a production compiler, we can evaluate the costs and benefits of generating a back end from machine descriptions, as compared with other methods of retargeting a compiler.

## 8.2 Ease of retargeting

I evaluate our process of generating a back end by comparing with the processes used to retarget the well-known retargetable compilers *lcc*, *vpo*,

<sup>2</sup>It may also be possible that the generated tileset incurs extra memory allocation, causing the compiler to spend more time in garbage collection. Unfortunately, Objective Caml does not have the necessary heap-profiling tools to properly pursue this question.

and gcc. Specifically, I evaluate how the compilers compare with one another according to the following criteria:

- *How much effort is required to build one back end?* In particular, how complicated are the algorithms implemented in the back end? If machine descriptions are used, how complicated is a machine description, and how complicated are the tools that manipulate the machine descriptions?
- *How much effort is required to build multiple back ends?* After a single back end has been built, what is the marginal effort required to add a new back end?
- *How much confidence do we have in the correctness of a new back end?* After a back end is added to the compiler, how do we verify that it generates correct code?
- *How much effort is required to add a new front end?* Specifically, what is the intermediate representation, and how well is the front end isolated from the code generator?

Ideally, I could use a quantitative measure to answer these questions, but it is notoriously difficult to quantify programming effort. Instead, I present descriptions, examples, and line counts to describe the retargeting process in each compiler.

Because the instruction selector is the majority of the machine-dependent code in a back end, we focus our discussion of the back ends on the implementations of instruction selection. Specifically, we study how a subtract instruction is emitted by each back end. The subtract instruction shows, without bringing in too much complexity, how a back end solves most of the problems in instruction selection. For example, on most machines the subtract instruction affects the condition codes, but it has few other complications. For some back ends, we need a more complicated instruction to highlight an interesting feature of the code generator. In these cases, we look at how the back end handles a multiply instruction.

### 8.2.1 Quick C--: hand-written back ends

A back end in Quick C--, whether hand-written or generated, uses the compiler structure described in Section 3.1.

**One back end** The code generator in Quick C-- consists of two components: the machine-dependent tileset and the recognizer. Because the tileset implements only a small number of simple tiles, most of the tileset code is short and simple. For example, on the *x86* we implement the tile for subtraction using the following Objective Caml function:<sup>3</sup>

```
let sub dst op x y = rtl (R.par
  [R.store (R.reg dst)
   (R.app (R.opr ("sub", [32]))
    [R.fetch (R.reg x); R.fetch (R.reg y)]) 32;
  R.store (R.reg eflags)
   (R.app (R.opr ("x86_subflags", [32]))
    [R.fetch (R.reg x); R.fetch (R.reg y)]) 32])
```

The main pitfall in implementing tiles is that it is easy to write code that usually works but may cause a bug in some corner cases. For example, our hand-written implementation of the *x86*'s shift instruction failed to specify an assignment to the condition codes, which could allow the optimizer to generate incorrect code.

The implementation of a hand-written recognizer is relatively simple because we rely on well-understood BURG technology (Fraser, Henry, and Proebsting 1992). Our implementation of the BURG engine is simple: it does not precompute state tables. The actual BURG specification has two parts: we write BURG patterns to match the machine instructions, and we write code to invoke the BURG engine on an input RTL.<sup>4</sup> The BURG pattern used to match the subtract instruction uses a constructor `Withaflags` to represent a binary operator that modifies the condition codes:

```
inst : Withaflags(dst:eaddr1, "sub", src:reg, w,
                  "x86_subflags")
      {: s "sub%s %s,%s" (suffix w) src dst :}
```

When the recognizer is invoked, the following pattern matches the subtraction RTL and constructs a BURG input tree that matches the pattern above, using the `conWithaflags` constructor:

<sup>3</sup>The given code is an inlined version of the actual code in the compiler. Because most of the binary-operator instructions on the *x86* are nearly identical, the actual compiler code is factored over the operator, allowing a shorter implementation of several operators.

<sup>4</sup>In the implementation, the invocation of the BURG engine is deforested. Instead of using data constructors to construct an input tree for BURG, we directly invoke lookup functions on the BURG state tables.

```

| RP.Rtl [(RP.Const(RP.Bool true),
          RP.Store(l, RP.App((o, _),
                             [RP.Fetch(l',_); r]), w));
         (RP.Const(RP.Bool true),
          RP.Store(RP.Reg(('c', _, _),
                    flag_index, _),
                  RP.App((fo, _), [RP.Fetch(l'',_);
                                   r']), _))]
] when l == l' && l == l'' && r == r'
      && flag_index == SS.indices.SS.cc ->
  conWithaflags (loc l) o (exp r) w fo

```

The code that matches RTLs and constructs BURG trees is not only verbose and tedious, but it is another possible source of tricky bugs: it is easy to write a pattern that does not check all of the preconditions of an instruction. For example, if the pattern above did not check that the first operand `l'` of the operation is equal to the destination `l`, then the compiler could generate code using an invalid machine instruction. Fortunately, this class of bugs is usually detected by the assembler.

The other major part of our hand-written back end is the handling of calling conventions and stack layout. We write a concise, formal specification of each in a domain-specific language that is interpreted by our compiler (Lindig and Ramsey 2004; Olinsky, Lindig, and Ramsey 2006; Dias and Ramsey 2006). The specifications are interpreted by fairly straightforward implementations of state machines. Writing a specification is no more complicated than understanding the conventions used by the target machine.

**Multiple back ends** Once a single back end has been implemented, we can reuse the recognizer's BURG engine and the domain-specific languages used for calling conventions and stack layout. But for each additional back end, we must write the following components from scratch:

- The code-generation tiles
- The recognizer's BURG patterns and the code that invokes the BURG engine
- The calling-convention and stack-layout specifications

**Testing** To test a back end, our hand-written back ends rely on end-to-end testing. Our test suite includes tests from our C front end and tests that have caused bugs in the past. We also have the mechanisms to generate tests for

our calling conventions, which can be used to verify that our compiler generates code that is interoperable with code generated by an existing compiler on the target machine.

**Front ends** Our compiler supports a variety of source languages by providing a flexible interface to front ends. The front end generates code in C--, which represents instructions using the full expressive power of the RTL language. With 90-odd operators, RTLs are capable of expressing a wide variety of machine-level computation, and new operators can be added easily.

Because RTLs describe instructions at the machine level, our intermediate language is not biased toward any particular source language. The flexibility of this approach is demonstrated by the variety of languages for which front ends have been written to target our compiler: C code with the `lcc` compiler, Standard ML with the `MLton` compiler, Java with the `Whirlwind` compiler, and the pedagogical language `Tiger` with a front end written by a colleague (Govereau 2003).

## 8.2.2 Quick C--: generated back ends

Our goal in generating a back end is to produce reliable compilers quickly, not to push automatic generation to its extreme. Some code should be automatically generated; some code could be generated, but might not be worth the effort; and some code cannot even be generated in principle.

We generate the machine-dependent tileset and recognizer because they are not only the most complicated parts of the back end but also the largest, comprising 67% of the back end, as measured in lines of code (see Table 8.4). The code in these components is also the most error-prone code in the back end: in the course of performing experiments with the generated back ends, we uncovered about half a dozen bugs in the hand-written versions.

**One back end** The machine descriptions used to generate a back end are as straightforward as the target machine. We describe the `x86` subtract instruction from our earlier example with the following  $\lambda$ -RTL code:<sup>5</sup>

---

<sup>5</sup> $\lambda$ -RTL has several features that help the machine expert factor common patterns in machine instructions, such as binary operations that set the flags. Over multiple instructions, this factoring allows us to define several machine instructions in very few lines of code, resulting in a compact description of the target machine. But for the purpose of comparing various compilers, we have highlighted the description of our example instruction by inlining the functions used in the  $\lambda$ -RTL description.

<b>Back-end component</b>	<b># lines</b>
Tileset	522
Recognizer	764
Stack layout	128
Calling conventions	73
Register specs	46
Calling convention glue	199
Back-end glue	179
<b>Total</b>	<b>1,911</b>

Table 8.4: Line counts for components in the hand-written x86 back end.

```

SUBmod (Eaddr, reg) is
  Eaddr      := sub (Eaddr, reg)
  | Reg.EFLAGS := x86_subflags(Eaddr, reg)

```

Compared to the hand-written tileset or recognizer, the machine description is short and simple: we encode the semantics of the machine instruction without concern for the data structures or algorithms used by the compiler. As a point of comparison, the SLED and  $\lambda$ -RTL machine descriptions use a total of 1,948 lines to describe 639 distinct instructions, whereas the hand-written tileset and recognizer require 1,286 lines of code and can generate only 233 distinct instructions. By generating a recognizer that accepts more instructions, our generated back end allows the optimizer to produce better code: the optimizer can only execute a transformation if the resulting RTLs are accepted by the recognizer.

Of course, the cost of implementing the tiler and the  $\lambda$ -RTL toolkit must be considered: the algorithms described in this dissertation are fairly complicated. The tiler is a complicated module that harbored lingering bugs throughout its development, but the formalization in this dissertation should make it significantly easier to ensure that a fresh implementation of the tiler can cover any input instruction. It is difficult to evaluate the effort that would be required to reimplement the algorithms in the  $\lambda$ -RTL toolkit because although the  $\lambda$ -RTL toolkit provided valuable infrastructure for manipulating declarative machine descriptions, much of the effort in extending the toolkit required changes to the existing, rather complicated, infrastructure. It is also not clear how much work would be required to use our implementation of these algorithms to generate code for more than one compiler (Ramsey 2000); much of the programming effort would surely be devoted to adapting

the algorithms to the compiler's particular representation of instructions; if the compiler does not use a similar form of RTLs, the implementation could be substantially different. But these algorithms are implemented only once, and a proper implementation should avoid the bugs introduced in hand-written expanders and recognizers.

The rest of a generated back end is unchanged from a hand-written back end. We could generate the list of available registers and their aliasing relationships, as discussed by Smith, Ramsey, and Holloway (2004). But because the code is short and hard to get wrong, we continue to write it by hand. Another 20% of the back end is glue code that constructs a record for each target machine, collecting various parts of the back end (e.g. the byte-order and wordsize of memory, the calling conventions, the expansion tiles, the recognizer, and information for the assembler). This large amount of code reflects the number of people who have had their fingers in that part of the compiler, and it is far more than necessary. It might be possible to generate this code, but although the size of this code is embarrassing, writing it is neither difficult nor time-consuming.

The remaining 13% of the code consists of the specifications of calling conventions and stack layout. We cannot generate the code for the stack layout and calling conventions for the target's standard C calling convention because both are a matter of human convention; they are not properties of the machine. But considering that we support five different calling conventions on the x86, the 201 lines of formal specification are reasonably short.

Although the standard calling conventions cannot be generated, we could probably generate native (non-standard) calling conventions automatically. We could then automate calling-convention experiments that others have performed by hand (Davidson and Whalley 1991) and generate a calling convention that performs well on a benchmark of choice.

**Multiple back ends** Once a single back end has been implemented, we can reuse the  $\lambda$ -RTL toolkit to generate tilesets and recognizers for new back ends. But for each additional back end, we must write the following components from scratch:

- Machine descriptions for the target machine
- The calling-convention and stack-layout specifications

**Testing** In addition to all the end-to-end tests used in our hand-written compiler, our generated compiler opens new possibilities for testing the

code generator. We can test the SLED machine description to verify that it correctly describes the encodings of machine instructions (Fernández and Ramsey 1997); in future work, we hope to develop similar mechanisms to test  $\lambda$ -RTL descriptions. Furthermore, we are interested in applying techniques from certified compilation (Leroy 2006) to prove that the generated tileset produces correct code. We believe that the inference rules for finding implementations of tiles can form the basis of a proof strategy.

**Front ends** When an automatically generated back end is used, the support for front ends remains unchanged.

### 8.2.3 `lcc`

The `lcc` compiler is a well-designed, portable C compiler that quickly generates reasonable, but mostly unoptimized, assembly code (Fraser and Hanson 1995). The only optimization in the compiler is common sub-expression elimination, and the compiler uses a local register allocator.

**One back end** A back end in `lcc` provides two major pieces: type metrics and a code generator. The type metrics describe the size and alignment of datatypes, which helps the front end lay out data. The code generator matches the compiler's intermediate representation and emits assembly code for the target machine. At the heart of the code generator is a BURG description mapping the compiler's intermediate representation to machine instructions. The advantages of using BURG are that it is easy to write a simple BURG engine (Fraser, Hanson, and Proebsting 1992), and it is easy to write BURG patterns that match the tree-based intermediate representation. For example, `lcc` uses the following BURG pattern to implement the subtract node in the compiler's intermediate representation:

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

Because the `x86` subtract instruction `subl` places its result in the same register as its first source operand, `lcc` emits a `movl` instruction to copy the source operand `%0` to the destination `%c`.<sup>6</sup> In many cases, the mapping from the intermediate representation to machine instructions is straightforward.

But the BURG description does not provide all the information about the selection instruction: in some cases, the BURG code may generate an instruction that requires its operands to be placed in specific registers. For

<sup>6</sup>The “?” prefix ensures that the move instruction is only emitted if the source and destination are different locations.

example, the *x86*'s multiply instruction requires one of the operands to be placed in the EAX register. To ensure that the register allocator places the operands in the proper locations, the back end must also provide a function target to ensure that the operands of an operator are placed in suitable locations. For example, the output of the multiply instruction must be the EAX register (defined elsewhere as quo), and the first input operand must be the EAX register.

```
case MUL+U:
    setreg(p, quo);
    rtarget(p, 0, intreg[EAX]);
```

As compilers go, an *lcc* back end is relatively simple, requiring only 939 lines of non-blank, non-comment lines for the *x86* back end. But the simplicity comes with a cost: the compiler does not support optimization on the selected machine instructions. The BURG engine promises only to produce locally optimal code, where optimality is determined by costs attached to the patterns in the BURG description.

More significantly, the effort required to add a new back end to *lcc* includes the effort of understanding *lcc*'s interface between the front end and the back end. Writing the type metrics is easy, and most instructions in the BURG description are relatively straightforward. But to understand complicated cases, such as the *x86* multiply instruction, we cannot reason locally about the BURG code. Instead, we have to understand that there is a separate target function that specifies where to allocate an operand in the intermediate code. Similarly, we have to know that the back end must define a clobber function to specify unintended side effects of the assembly code.

The *lcc* approach contrasts strongly with our approach of generating the code generator from declarative machine descriptions. Our generated compilers support optimization on machine instructions by following Davidson's code-generation strategy. And we achieve a separation of concerns by requiring only declarative descriptions of the target machine: we can describe the instructions on the target machine without understanding how the compiler will use the generated components.

**Multiple back ends** Once a single back end has been implemented, we can reuse the BURG engine. But for each additional back end, we must write the following components from scratch:

- Type metrics
- BURG code mapping intermediate code to machine instructions
- Additional functions constraining register use in instructions

**Testing** To test a new back end, the `lcc` compiler relies on end-to-end testing. The compiler has been tested on the Plum-Hall Validation Suite for ANSI C compilers, in addition to the compiler's custom test suite, including previously reported bugs.

**Front ends** Although it is easy to add a new back end to `lcc`, it is not easy to add a new front end. The front end and the back end are tightly coupled: the interface consists of 19 functions with callbacks made in both directions across the interface. Data structures are also shared across the interface, most notably symbols and types. The intermediate representation uses directed acyclic graphs with a 36-operator language to represent instructions. The types and operators are precisely those needed for a C compiler, which may make it difficult to shoehorn another source language into the given intermediate representation. This design contrasts strongly with C--, which provides a broad range of operators and lets the front end make the decisions about data representation. Fraser and Hanson (1995) acknowledge these limitations while explaining that adding front ends easily was not a design goal for `lcc`, "changes to front end may affect back ends... This complication is less important for standardized languages like ANSI C because there will be few changes to the language." Unfortunately, even standardized languages change. Despite the quality of `lcc`'s design and implementation, it has been rendered obsolete by changes to both the de jure C standard (C99) and the de facto C standard (`gcc`).

## 8.2.4 vpo

The `vpo` compiler is a highly optimizing compiler designed to be ported easily to new target machines. With a collection of scalar and loop optimizations, `vpo` demonstrates that a Davidson/Fraser compiler can produce extremely fast code (Benitez and Davidson 1994).

**One back end** The main components of a `vpo` back end are the expander and the recognizer. The expander is a combination of our tiler and machine-dependent tileset: it is a single machine-dependent component that takes an intermediate representation of the source program as input and returns semantically equivalent RTLs, each of which is representable by a single instruction on the target machine. For example, the following case expands a subtract instruction in a manner very similar to our hand-written tileset:<sup>7</sup>

---

<sup>7</sup>An extra case for floating-point has been elided to simplify the exposition.

```

case 0_SUB:
    src2 = pop_loc();
    src1 = pop_loc();
    src2_size = get_location_size(src2);
    src1_size = get_location_size(src1);
    VP0i_rtl(sub(src1, Rtl_fetch(src2,src2_size*8), src1_size),
             VP0i_locSetBuild(EFLAG,src2,0));
    push_loc(src1);

```

The VP0i\_rtl function outputs an RTL and a set of registers killed by the instruction. In this case, the condition-code register EFLAG is killed, as is the source operand. The RTL representing the subtract instruction is constructed by the sub function:

```

Rtl_ty_rtl sub(Rtl_ty_loc reg_or_mem,
              reg_or_immInstance reg_or_imm, int nbytes) {
    Rtl_ty_expr expr;
    expr = Rtl_binary(Rtl_op_sub,
                    Rtl_fetch(reg_or_mem,nbytes*8), reg_or_imm);
    return Rtl_assign(reg_or_mem, nbytes*8, expr); }

```

As in our hand-written tileset, the vpo expander constructs the familiar subtract instruction on the x86, which stores the result in the first source operand.

As in our compiler, vpo's recognizer implements a predicate that decides whether an RTL is representable by a single instruction on the target machine; the recognizer can also return assembly code for the instruction. Because vpo represents an RTL as a string, vpo's recognizer uses a parsing engine, YACC, to match instructions. The implementation of the parsing engine requires some effort, but techniques for parser generation are well known, and existing implementations are ubiquitous. Once the parsing engine is written, the compiler writer creates a grammar that matches valid RTLs for the target machine. For example, the following YACC patterns match the x86 subtract instruction:

```

inst  : subi                {binst($1, SN);}
subi  : dst '=' rhssub ';'  {$$ = binopi($1, $3);}
rhssub: src1 '-' src2      {$$ = brecord('-', $1, $3, AN);}

```

But these patterns also match a subtract instruction in which the destination is not the same as the first source operand, which is not a valid instruction on the x86. The recognizer uses a library of functions to check the validity of the instructions that are parsed by the grammar. For example, the action of the subi pattern invokes the function binopi, which checks that, among other conditions, the destination matches the first source operand.

Because a YACC pattern may match several instructions, the conditions in a recognizer function may be very complicated: the function `binopi` is 58 lines of code.

Overall, `vpo`'s recognizer is very similar to our hand-written recognizer: both recognizers use matching technology to facilitate the task of writing instruction patterns, and both recognizers require a raft of complicated condition-checking to ensure that the recognizer accepts only valid RTLs.

Like our hand-written back end, the `vpo` back end is significantly more complicated than a declarative machine description. In total, `vpo`'s expander for the `x86` consists of 922 non-blank, non-comment lines of code, and the recognizer consists of 1,840 lines. But more importantly, the code is complicated: the expander must select machine instructions for any C program, and the recognizer requires not only a YACC description to parse the RTLs but also over 50 helper functions consisting of 1,502 non-blank, non-comment lines of code to verify that the RTLs are actually valid on the target machine.

**Multiple back ends** Aside from the parsing engine, there are no reusable parts in the code generator. For each additional back end, the expander and recognizer must be written by hand.

**Testing** To test a new back end, `vpo` uses end-to-end testing with an extensive test suite. Additionally, the idea of specifying and generating tests for calling conventions was pioneered by `vpo` (Bailey and Davidson 1995).

**Front ends** In `vpo`, adding a new source language requires work proportional to the number of target machines we want to support. The front end in `vpo` is expected to convert the source program to `vpo`'s intermediate representation: RTLs with the invariant that each RTL is representable by a single instruction on the target machine. The conversion is performed by the expander, which therefore must be both specific to the source language and the target machine. The consequence is that support for a new language requires a separate expander for each target machine.

### 8.2.5 `gcc`

The `gcc` compiler is a widely used, highly optimizing compiler. An army of contributors have implemented a variety of scalar and loop optimizations, as well as many different back ends. The result is a compiler that generates good code on many different machines. The compiler has evolved significantly over time; in this discussion, I describe version 4.2.1.

**One back end** The structure of `gcc` is a mixture of the canonical dragon-book compiler and Davidson's code generation strategy. Each front end generates code in a (mostly) language-independent, machine-independent tree-based intermediate representation. The tree-based representation is converted to SSA form, and a number of optimizations are performed. Then, the code generator takes over.

The code generator is based on Davidson's structure of a compiler: the code is converted to a form of RTLs using an expander, and the machine invariant is maintained by a recognizer. But in `gcc`, the expander and recognizer are not written by hand. Instead, the compiler writer defines a "machine description," from which the expander and recognizer are generated.

But `gcc`'s machine descriptions are not like  $\lambda$ -RTL machine descriptions. A `gcc` machine description is a collection of definitions written in a domain-specific language designed to implement the expander, the recognizer, and sometimes parts of the optimizer. For example, the expansion of a subtract instruction is defined as follows on the `x86`:

```
(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;")
```

The first argument "subsi3" states that the code expands a subtract instruction that works on 4-byte integers. The second argument is an RTL with two effects: one performing the subtraction and the other clobbering the condition-code register. The careful reader will note that the RTL does not constrain the destination location to be the same as the first source operand. But strangely, it appears that this RTL is never used. Instead, the final argument of the `define_expand` definition contains arbitrary C code that is evaluated before generating the expanded RTL. In this case, the `ix86_expand_binary_operator` function emits the correct RTL, with the destination register properly constrained. Finally, the `DONE` macro ensures that the RTL emitted by `ix86_expand_binary_operator` is used instead of the RTL in the second argument.

The recognizer uses a similar definition to describe machine instructions. The following definition gives a pattern that matches the subtract instruction:

```
(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{1}\t{1,%2, %0|%0, %2}"
  [(set_attr "type" "alu") (set_attr "mode" "SI")])
```

The first argument `"*subsi_3"` is an optional name for the pattern. The second argument is an RTL pattern that matches the subtract instruction. The third argument contains arbitrary C code, which in this case invokes the predicates `ix86_match_ccmode` and `ix86_binary_operator_ok`, which perform additional checks on the matched RTL. For example, the latter predicate verifies that the destination of the RTL is the same as the first source operand. The final two arguments specify the assembly code string for the instruction and some optional attributes.

The contrast between declarative machine descriptions and gcc's machine descriptions cannot be overstated. Whereas a declarative machine description is designed to concisely describe a property of a machine, independent of any particular tool, the domain-specific language used by gcc is verbose and requires a knowledge of gcc-specific conventions. For example, the code for expanding a subtract instruction uses the gcc-specific tree constructors (e.g. `match_operand:SI`) to describe the instruction. More significantly, the snippets of arbitrary C code are fundamental to gcc's retargeting strategy. Rather than develop a complete representation of the instruction set that can be analyzed and understood independent of the machine, the compiler writer is expected to write critical parts of the machine-dependent code by hand (e.g. predicates for matching RTLs) and rely on gcc's macro framework to include the hand-written code in the generated instruction selector.

In total, gcc's machine description for the x86 requires 18,462 non-blank, non-comment lines of code to define 754 instructions, 320 expansions, and 83 machine-specific peephole optimizations. The gcc machine description is an order of magnitude longer than the  $\lambda$ -RTL machine description.

**Multiple back ends** The code that generates the expander and recognizer is complicated, but it is written only once, then reused with all the compiler's

back ends. The code to generate an expander is 727 non-comment, non-blank lines of C code; the code to generate a recognizer is 2,289 lines of C code.

Once a single back end has been implemented, the additional effort required to add a new code generator is writing a new gcc machine description:

- Definitions of the machine instructions
- Definitions for the expander

**Testing** To test a new back end, gcc relies on end-to-end testing, using an extensive test suite.

**Front ends** In addition to supporting a large number of back ends, gcc can support a new front end through an interface that is only partially decoupled from the rest of the compiler. The front end is responsible for converting a source program into code in a language-independent, machine-independent tree-based intermediate representation. gcc then translates the tree-based intermediate representation into another internal representation called GIMPLE. Unlike C--, the tree-based intermediate representation was designed for C programs, making it well-suited to procedural languages but not necessarily other languages. If a front end cannot compile a language feature into the tree-based representation, the compiler writer has one recourse: he can describe the feature using a language-dependent extension to the tree-based representation and provide callback functions to convert the extension into GIMPLE. In short, if the tree-based representation is not expressive enough to capture the semantics of the source language, the writer of the front end can use a mechanism to extend the early phases of the compiler. Despite the effort that may be involved in adding a front end for a non-procedural language, gcc supports front ends not only for C and C++, but also for the object-oriented language Java and the hybrid logic-functional language Mercury.

## 8.3 Summary

In this chapter, we have evaluated the back ends we generate automatically, in comparison with well-known retargetable compilers. To add a new target to the other compilers, we must write a translation from the compiler's intermediate representation to machine instructions. Each compiler provides support in the form of a domain-specific language (BURG for lcc,

YACC for vpo, and custom macros for gcc). But even in a domain-specific language, the code for each instruction is more complicated than a simple description of the instruction's semantics, often requiring additional code to be written in C because it cannot be expressed in the domain-specific language.

Unlike the other compilers, our approach requires a large amount of infrastructure (i.e. the  $\lambda$ -RTL toolkit) to analyze the machine descriptions and generate the back ends. But once implemented, the infrastructure can be reused to generate new back ends with a minimal amount of effort: if we describe the syntax and semantics of the target instruction set, the  $\lambda$ -RTL toolkit can produce the code generator. Our experiments show that, unlike previous attempts to generate the back end of a compiler (Chapter 9), our generated back ends are as good as hand-written back ends, suitable for use in a production compiler.

# Chapter 9

## Related Work

Ever since UNCOL (Strong et al. 1958), people have been working on retargetable compilers. Instead of surveying all that work, we discuss three kinds of related work: the construction of retargetable compilers, the design of machine-description languages, and the automatic generation of compiler back ends.

### 9.1 Approaches to retargeting compilers

In Chapter 8, we discussed how `lcc`, `gcc`, `vpo`, and `Quick C--` can be ported to a new platform. The `lcc` compiler provides an almost ideal example of how a dragon-book compiler can be ported with minimal effort. The other compilers all demonstrate how a Davidson/Fraser compiler can be ported to a new machine while providing machine-specific optimization.

But another notable approach was developed in the Machine SUIF compiler (Smith 1996), and while it is similar in spirit to a Davidson/Fraser compiler, the execution is quite different. The key difference is the representation of machine instructions. In a Davidson/Fraser compiler, the representation of an instruction is *manifest*: every compiler component knows the instruction is represented using an RTL, and each compiler component directly analyzes and modifies the RTLs. In Machine SUIF, the representation of an instruction is *abstract*: each back end may have a different representation, and the back end implements an interface that provides functions for analyzing the instructions. For example, each target must provide a function that returns the set of registers used in an instruction. The compiler components are then able to manipulate the machine instructions through the use of the functions in the interface. Like a Davidson/Fraser compiler, this architecture admits machine-level optimizations that are written in a machine-independent fashion. But unlike a Davidson/Fraser compiler, the

amount of work required to retarget the compiler is not entirely independent of the optimizations: if an optimization requires a new function in the instruction interface, then the function must be implemented in every back end. For example, if you want to write an optimization to take advantage of prefetching, each target may be extended with a predicate that identifies prefetching instructions.

## 9.2 Machine descriptions

Many different kinds of things are called “machine descriptions,” but too often the term is used loosely to mean “whatever information is needed to retarget my compiler.” Most descriptions are written in domain-specific languages, which are designed with the singular goal of generating a code generator. The goal of my work is to eliminate the need for these domain-specific languages in favor of reusable machine descriptions, which are designed to support analyses for generating a variety of tools, not just compilers.

### 9.2.1 Domain-specific languages

In some cases, the term “machine description” is used to describe a domain-specific language that maps a compiler’s data structures to machine instructions. For example, a BURG specification (Fraser, Henry, and Proebsting 1992) describes a mapping from the compiler’s intermediate representation to machine instructions. Similar specifications have been used in other code-generator generators (Glanville and Graham 1978; Aho, Ganapathi, and Tjiang 1989; Farfeleder et al. 2006), and the generator BEG adds support for generating a register allocator using information about register sets (Emmelmann, Schröer, and Landwehr 1989). Because these tools map the compiler’s intermediate representation to machine instructions, the description cannot be analyzed independent of the compiler. But even worse, the descriptions contain arbitrary fragments of C code, which cannot be analyzed. The result is that the machine descriptions are useful only for one purpose: generating a code generator.

Like BURG descriptions, the machine descriptions used by gcc and vpo relate the compiler’s intermediate code to machine instructions. Both compilers use the term machine description to refer to their definitions of the expander and the recognizer. Because the code is specific to the compiler and its intermediate representation, the machine descriptions are neither analyzable nor reusable .

## 9.2.2 Reusable languages

Reusable machine descriptions are designed to describe the properties of a machine, such as syntax or semantics, with the goal of admitting analyses for generating machine-specific tools. The bulk of the work on machine descriptions has taken place in the design automation and embedded communities, with little cross-fertilization with the compiler community.

One of the first machine-description languages that was intended to be reusable was ISP (Bell and Newell 1971). In ISP, a description defined the storage locations on the machine and an interpreter, which decoded and executed the machine instructions. One of the results of the project was the observation that non-declarative specifications are difficult to reuse. Consequently, Oakley (1979) developed a method to extract a declarative description of the instructions from the ISP interpreter.

Like ISP, modern machine-description languages, such as LISAS (Cook et al. 1993), nML (Fauth, Praet, and Freericks 1995), TOAST (Hoover and Zadeck 1996), ISDL (Hadjyiannis, Hanono, and Devadas 1997), SLED (Ramsey and Fernández 1997),  $\lambda$ -RTL (Ramsey and Davidson 1998), and LISA (Braun et al. 2004), consist of two parts: a specification of the storage space and a specification of the instructions. Although each language has a unique syntax, type system, and model for describing instructions, each of the instruction descriptions can be understood as defining the assignments to the machine state. I believe that each instruction specification could be mapped onto the RTLs defined in a  $\lambda$ -RTL description, which means that I could adapt the work in this dissertation to generate compilers from machine descriptions written in these languages.

But don't think that all machine descriptions are equal. The syntax and the model of instructions are crucial to writing concise, readable (and hence debuggable) machine descriptions. A machine description written in TOAST, for example, is extraordinarily verbose when compared with a  $\lambda$ -RTL machine description. TOAST, on the other hand, has the ability to describe families of machine descriptions, which is a feature sadly lacking in  $\lambda$ -RTL.

Another important consideration is the expressive power of the language. In LISAS (Cook et al. 1993), an addressing mode may define not only how an operand is fetched but also a side effect on an addressing register; this side effect is then propagated *implicitly* to any instruction that uses the addressing mode. The order in which the side effect takes place relative to the other assignments of the instruction (before, in parallel, or after) is determined by textual order in the machine description (Cook et al. 1993, Section 3.2), which may make specification of some instructions difficult or impossible.

### 9.3 Generating code generators from machine descriptions

A variety of projects have worked on generating compiler back ends. Because BURS-style and gcc-style machine descriptions are really domain-specific languages for defining compiler components, the tools that generate the compiler components are not particularly interesting: the human who writes the domain-specific descriptions has already done the interesting work of analyzing the instruction set. We focus instead on projects that have tried to generate code generators using analyses of machine descriptions that are intended to describe the machine independent of the compiler.

At the heart of generating a code generator is the problem of proving that a sequence of machine instructions is equivalent to some other sequence of instructions. Early work on generating optimizers (Kessler 1986) focused on finding a sequence of machine instructions that was equivalent to a single machine instruction. An optimization pass could then replace a matching sequence in a program with the single equivalent instruction. Kessler's work relied mostly on syntactic matching rather than semantic matching using algebraic laws.

Another important project from the optimization community is the superoptimizer (Massalin 1987; Granlund and Kenner 1992), which searches for sequences of instructions that can efficiently implement a given operation. Because the superoptimizer conducts an exhaustive search for sequences of instructions, it finds efficient, and often surprising, code improvements. But because of the expensive nature of the search strategy, it is not clear how the superoptimizer might be adapted to find implementations of expansion tiles. Furthermore, the superoptimizer does not stand alone: it must be able to test candidate sequences, either by using the target machine or by using an emulator.

A subsequent project named Denali showed how a superoptimizer can be built using a theorem prover (Joshi, Nelson, and Randall 2002; Joshi, Nelson, and Zhou 2006). The theorem prover searches for a sequence of machine instructions that implement an input goal procedure. Because the theorem prover only produces semantically correct results, Denali does not need to test the implementations it produces. Early results show that Denali can find high-quality implementations of register-to-register computations.

Cattell's early work on generating an instruction selector (Chapter 7) took place in the context of the PQCC project, which was tasked with developing compiler-construction techniques that isolate machine-dependent components so that they can be generated automatically (Lunell 1983). Cat-

tell's work focused on generating the code generator using a machine description derived from ISP (Bell and Newell 1971).<sup>1</sup> Like other work on generating a back end, Cattell uses a compiler where instruction selection is one of the final phases, so instead of being able to generate naïve code and improve it later, Cattell's automatically generated instruction selector had to generate good code. Unfortunately, experimental results are not available.

In the TOAST project, Hoover and Zadeck (1996) developed a machine-description language and an algorithm for generating instruction selectors. Kessler's work was identified as an early influence on TOAST (Hoover and Zadeck 1996), but the published algorithm appears more similar to Cattell's work. The most notable difference is that they restrict the use of algebraic laws to ensure a finite search space, which they search exhaustively. They describe their search in terms of *toeprints* and *footprints*:

- A *toeprint* is a single assignment in a machine instruction that computes a “desired operation,” where a “desired operation” in my terminology would be either the expression computed by an expansion tile or a subexpression thereof. A toeprint is discovered through a means-ends search in the style of Cattell, complete with the use of algebraic laws to direct the search.
- A *footprint* is the combination of any number of toeprints from the same instruction, which is essentially the parallel composition of the potentially useful assignments that may be executed by a single instruction.

TOAST's instruction selector uses the footprints as tiles to cover the compiler's intermediate representation. In addition to generating the instruction selector, another goal of the TOAST project was to generate machine-specific optimization passes. Unfortunately, we have not been able to find experimental results from the code generators produced by TOAST.

Much of the work on generating code generators in embedded systems (Lanneer et al. 1995; Hanono and Devadas 1998) has focused on the difficult problem of efficient tiling in a VLIW architecture, while ignoring the question of how to extract a sufficient set of tiles from the machine descriptions.

---

<sup>1</sup>Cattell's dissertation states that Oakley's work (Oakley 1979) on automatically deriving the machine description from an ISP description was not yet finished, so Cattell “manually generated” his machine descriptions (Cattell 1982, Section 2.6, pg 22). In a journal paper, Cattell suggested that he used machine descriptions automatically generated by Oakley, but the results section described the amount of time required to write the machine descriptions by hand (Cattell 1980).

A notable exception is recent work with LISA that discussed how to generate a BURG-style instruction selector using a LISA machine description (Ceng et al. 2005). The code produced by the generated back end ran about 5% slower than the code produced by a hand-written back end. This slow-down is indicative of the difficulty of generating a back end when the instruction selector is one of the last phases of the compiler: inefficient code produced by the instruction selector results in an inefficient program. Our novel approach, which builds on the Davidson/Fraser compilation strategy, allows the optimizer to clean up the inefficient code produced by our generated expander.

Another notable project retargets a compiler by exploiting an existing C compiler. Collberg (1997) uses the existing C compiler to generate code for some small sample programs, then analyzes the assembly and object code produced by the compiler to develop a model of the machine. Collberg's tool can then generate a description of the architecture for use with a tool like BEG, which can then generate a back end. Of course, this approach has limitations: it can only discover facts about the architecture that are exposed by the native C compiler, and it may not be amenable to handling code on particularly complex architectures. Nonetheless, it is amazing that the bear dances at all.

An important limitation across all these projects is that they attempt to generate an instruction selector for a dragon-book compiler, with the consequence that any inefficient code produced by the instruction selector is present in the compiled programs. Our work avoids this problem by adopting the Davidson/Fraser compilation strategy: after our instruction selector chooses code, the optimizer can improve the code.

# Chapter 10

## Conclusions

In this dissertation, I have shown how to automatically generate back ends for an optimizing compiler. By generating the machine-specific components of the compiler, I enable a separation of concerns:

- The language expert can write a front end to compile his language to C++ and get the performance of native code without the effort of compiling to assembly code or to C.
- The compiler expert can write target-independent optimizations that improve machine instructions without the effort of writing new optimizations for each target.
- With only a superficial knowledge of the compiler, the machine expert can add a new target machine: he writes a declarative machine description along with a few short descriptions of platform-specific conventions.
- An expert in both the compiler and the machine can write a machine-specific optimization designed to exploit architectural features of a specific target machine. The same optimization may apply to other machines that have the same architectural features.

Our compiler builds upon the work of Davidson and Fraser (1984), who showed how to write a compiler by selecting instructions early, then using a machine-independent optimizer to improve the machine instructions. They also showed that machine-dependent optimizations can be written in the machine-independent optimizer, then applied to multiple machines. We adapt the structure of a Davidson/Fraser compiler to divide the expander into a machine-independent tiler and a machine-dependent tile set. The tiler

allows us to divide the translation from the source language to machine instructions into two steps: the language-dependent front end generates C--, and the tiler chooses machine instructions. The tiler plays the same sort of role as gcc's tree language, but because the tiler can handle any well-typed, machine-sized RTLs, it does not entail any loss of expressive power.

## 10.1 Summary of results

I have implemented the algorithms described in this dissertation in the  $\lambda$ -RTL toolkit, which can now generate new back ends for the Quick C-- compiler. I have generated back ends for the x86, PowerPC, and ARM architectures, and experiments show that the back ends are of suitable quality for a production compiler. Unlike previous work on generating back ends, we rely on the optimizer to improve the naïve code produced by our generated components. The result is that a compiler using our generated back ends can produce code that is as good as the code produced by a compiler with a hand-written back end.

## 10.2 Contributions

One technical contribution of this dissertation is the automatic generation of a recognizer. Using the analyses developed by Feigenbaum (2001), the generated recognizers accept the inverse image of the instruction set, where the transformation in question is a sequence of passes in the compiler. The generated recognizers are efficient, BURG-style matchers, although the choice of matching technology is relatively unimportant: any match compiler that generates efficient, small matchers would suffice. More importantly, the generated recognizers are produced with less effort and less risk of human error. And because it is easier to add instructions to a machine description than it is to add instructions to a hand-written recognizer, the generated recognizers accept more machine instructions than the hand-written versions.

The primary technical contributions of this dissertation are the proof that generating an instruction selector is undecidable (Appendix A), the novel algorithm for finding implementations of intermediate code, and the heuristic I use to prune the search. Although the undecidability result seems obvious in retrospect, I am unaware of any mention of undecidability in previous work on generating back ends. Because there can be no decision procedure, our algorithm for finding implementations of expansion tiles has to use heuristic search.

A novel contribution of this dissertation is the use of forward chaining to search for implementations of expansion tiles. The search constructs a sequence of machine instructions with the goal of establishing a state in which an algebraic law may be applied, leading to the discovery of an implementation of a new instruction. Unlike previous work that used forward chaining (Cattell 1982; Hoover and Zadeck 1996), the novel invariant that guides my search is to construct only sequences that consist of valid instructions on the target machine. This invariant is a powerful restriction on the search space that has the effect of tailoring the search to the target machine.

To further restrict the search, I developed a novel pruning heuristic that estimates whether an implementation of an instruction is likely to be used in implementing an expansion tile. If not, we can discard the implementation, limiting the size of the search space. The heuristic estimates the number of algebraic laws that must be applied to an instruction before it may possibly yield an implementation of a tile. This heuristic is based on the postulation that an implementation of an expansion tile is more likely to be found with the application of a small number of algebraic laws. The combination of this pruning heuristic with my forward-chaining search strategy is an algorithm that generates an instruction selector in a matter of minutes.

### 10.3 Future work

In future work, I hope to extend this dissertation to prove that the back ends I generate produce correct code. Because a machine description can be analyzed, it should be possible to write a tool that checks the correctness of the machine description. Previous work has already shown how to check the correctness of SLED descriptions (Fernández and Ramsey 1997); in future work, I want to devise an algorithm for checking whether a  $\lambda$ -RTL description is an accurate description of the target machine.

Furthermore, I would like to prove that each tile in a generated back end produces code that preserves the semantics of the intermediate representation. There are three main approaches to building provably correct compilers:

- *Translation validation*: Run the tiler and check whether it produces code that maintains the input program's semantics (Pnueli, Siegel, and Singerman 1998; Necula 2000).
- *Proof-carrying code*: Along with each tile in the generated tiler, include a proof that the tile's implementation preserves the semantics of the intermediate code (Necula 1997; Morrisett et al. 1999).

- *Compiler certification*: Prove that the code that generates the tile set can only generate semantics-preserving tiles (Leroy 2006).

For my purposes, I believe that proof-carrying code is the most practical next step because unlike compiler certification, it does not require rewriting the  $\lambda$ -RTL toolkit in a proof checker, and unlike translation validation, the generated tile set can be checked for correctness independent of any single test program. I hope that I will soon be able to generate back ends for an optimizing compiler that are not just efficient but also provably correct.

## Appendix A

# Automatically generating an instruction selector is undecidable

Despite numerous attacks on the problem over many years, automatically generating an efficient instruction selector from machine descriptions has remained a problem without a definitive solution. As it turns out, the general form of the problem is not just difficult, it is undecidable: there is no terminating algorithm that is guaranteed to find an implementation of a code-generation tile if one exists. In retrospect, this undecidability result is unsurprising: the problem of finding a sequence of machine instructions that are equivalent to a code-generation tile is closely related to the more general, undecidable problem of determining if two programs are equivalent.

In fact, even the special case of deciding whether two machine instructions are equivalent under a set of algebraic laws is undecidable. In this chapter, we give the proof of undecidability for this special case, which is sufficient to show that the general problem is undecidable. Before we can give the proof of undecidability, we need a clear definition of the problem.

### A.1 The Instruction Equivalence Problem

The problem is to decide whether one instruction  $i_1$  is equivalent to another instruction  $i_n$ , under some set of algebraic laws. To validate correctness, the algorithm must provide a proof of the equivalence consisting of a sequence of steps  $i_1 \Rightarrow i_2 \Rightarrow \dots \Rightarrow i_n$ , where each step  $i_k \Rightarrow i_{k+1}$  applies a single algebraic law to conclude that  $i_k$  and  $i_{k+1}$  are equivalent. If we view each step of the sequence as rewriting the instruction, then we lose no expressive power by replacing the algebraic laws with a set of rewrite rules;

an algebraic law  $x = y$  can be rewritten as a pair of rewrite rules  $x \Rightarrow y$  and  $y \Rightarrow x$ . Now I can give a more formal definition of the problem.

**Definition 1.** *Instruction Equivalence Problem:* Given two instructions  $i_1$  and  $i_n$  (represented as RTLs for convenience) and a set of rewrite rules  $\mathcal{R}$ , the algorithm must determine whether the instructions are equivalent under  $\mathcal{R}$ , and if so, it must return the proof of equivalence as a sequence of single rewrites  $i_1 \Rightarrow \dots \Rightarrow i_n$ .

It is worth noting that my definition of the problem requires the algorithm to compute a solution that relies on algebraic reasoning. Hence, my proof of undecidability does not hold for algorithms that do not rely on algebraic reasoning. For example, an algorithm relying on exhaustive evaluation of two instructions on all possible machine states is not considered under my definition of the problem. Of course, exhaustive evaluation is unthinkable as a general solution: even a simple addition instruction with two 32-bit operands must consider  $2^{64}$  possible combinations of input values, and the state space would be even worse if we considered sequences of instructions. But more importantly, all the previous work on generating an instruction selector that I am aware of has been based on some form of algebraic reasoning to generate an implementation of the instruction selector.

## A.2 Proof of undecidability

The proof of undecidability proceeds by reduction from the halting problem, adapting well-known proofs of undecidability for strong normalization in term-rewriting systems (Huet and Lankford 1978; Klop 1992; Bezem, Klop, and Roel de Vrijer 2003). In particular, I have adapted much of the formalism from Bezem, Klop, and Roel de Vrijer (2003, Section 5.3, pp 152–155).

The proof strategy is to show an embedding of the Turing machine into the Instruction Equivalence Problem such that an algorithm solving the Instruction Equivalence Problem finds an equivalence between its representations of the starting state and the halting state if and only if the Turing machine will halt when given the same starting state. Because the halting problem is undecidable, the Instruction Equivalence Problem must also be undecidable.

### A.2.1 Defining a Turing machine

I borrow my definition of a Turing machine from Bezem, Klop, and Roel de Vrijer (2003):

**Definition 2.** A deterministic Turing machine is defined by a triple  $\langle Q, S, \delta \rangle$ , where

- $Q$  is a set of states  $\{q_1, \dots, q_n\}$
- $S$  is the set of symbols  $\{s_1, \dots, s_n, \square\}$  that may be written on the tape, and  $\square$  is the blank symbol;  $Q \cap S = \emptyset$
- The transition function  $\delta$  is a partial function of type  $Q \times S \rightarrow Q \times S \times \{L, R\}$ .  
The names  $L$  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 three values:

- The new state  $q'$  of the machine
- The new symbol  $s$  to write at position  $p$
- The direction to move along the tape: left or right

I represent the tape in the Turing machine as a triple: a stack that contains the symbols to the left of the current position, the symbol at the current position, and a stack that contains the symbols to the right of the current position. A stack  $T$  is represented using the following notation:

$$T ::= \circ \quad \text{blank stack} \\ \quad | \quad ST \quad \text{symbol } S \text{ followed by a stack } T$$

Sometimes we use a metavariable  $T$  to name a stack, with  $T_l$  and  $T_r$  frequently used to refer to the left and right stacks in a Turing machine. The blank stack stands for an infinite stack of blank symbols, which is used to represent the infinite extension of the tape in either direction. For example, the triple  $(s_1 s_2 s_3 \circ, s_4, s_5 s_6 \circ)$  stands for the tape where the symbol to the left of the current position is  $s_1$ , followed by  $s_2$ ,  $s_3$ , and a blank stack; the current symbol is  $s_4$ ; and the symbol to the right of the current position is  $s_5$ , followed by  $s_6$  and a blank stack.

We describe the configuration of the Turing machine at any given point using a pair consisting of the current state and the tape. For example, if the machine is in the state  $q_3$  and the tape is the same as our previous example, the configuration is  $\langle q_3, (s_1 s_2 s_3, s_4, s_5 s_6) \rangle$ . If the transition function induces a transition between two configurations  $c_1$  and  $c_2$ , we represent the transition using the notation  $c_1 \rightarrow c_2$ .

To reduce the halting problem to the Instruction Equivalence Problem, we define an embedding from a Turing machine configurations and transitions to machine instructions and rewrite rules  $\mathcal{R}$ . The one complication of this strategy is that we must be able to identify both the initial state and the halting state precisely in order to invoke an algorithm that solves the Instruction Equivalence Problem. The initial state is well known, but there may be many halting states. The solution I adopt is to ensure that the embedding defines a distinguished halting state and allows any halting state to be rewritten to the distinguished halting state.

### A.2.2 Encoding Turing-machine states as instructions

For each state  $q \in Q$ , we define a 3-argument operator  $q$  on the machine. For each symbol  $s \in S$ , we define a unary operator  $s$  on the machine. Furthermore, we use the literal 0 to represent an infinite number of blank symbols, and we use the literal 1 as a dummy parameter for the current symbol. We also define a fixed location  $l_0$  on the machine. Now, we can define a function  $\phi$  to embed a Turing-machine configuration into an instruction:

$$\begin{aligned}\phi(\langle q, (T_l, s, T_r) \rangle) &= l_0 := q(\phi_T(x), s(1), \phi_T(y)) \\ \phi_T(\circ) &= 0 \\ \phi_T(ST) &= S(\phi_T(T))\end{aligned}$$

For example, the embedding of  $\langle q_3, (s_1 s_2 s_3, s_4, s_5 s_6) \rangle$  produces the instruction

$$l_0 := q_3(s_1(s_2(s_3(0))), s_4(1), s_5(s_6(0))).$$

We also define an instruction to represent a distinguished halting state:  $l_0 := \text{halted}()$ , where  $\text{halted}$  is a new nullary operator. Our reduction must ensure that the embedding of a halting state in the Turing machine can be rewritten to this instruction.

Now, we define the embedding of the transition function in the Turing machine into a set of rewrite rules. The embedding is complicated by the fact that the literal 0 in our embedding really stands for an infinite stack of blank symbols. Consequently, for each transition in the Turing machine, we have to add two rewriting rules: one for a blank stack and one for a non-blank stack.

For each transition  $\delta(q, s) = (q', s', L)$  to the left, we add two rewrite rules. The first rule is used when we know at least one symbol on the stack to the left:

$$l_0 := q(\mathcal{S}_x(\mathcal{T}_l), s(1), \mathcal{T}_r) \Rightarrow l_0 := q'(\mathcal{T}_l, \mathcal{S}_x(1), s'(\mathcal{T}_r))$$

Each metavariable used in a rewrite rule is a capital letter in a calligraphy font. This rule uses metavariables  $\mathcal{T}_r$  and  $\mathcal{T}_l$  to stand for the expressions representing the left-hand and right-hand stacks in the machine configuration. The rule also uses a metavariable  $\mathcal{S}_x$  to stand for the operator representing the symbol on top of the left-hand stack.<sup>1</sup>

To handle the case when the left-hand stack is blank, we add a second rule for the transition  $\delta(q, s) = (q', s', L)$ :

$$l_0 := q(0, s(1), \mathcal{T}_r) \Rightarrow l_0 := q'(0, \square(1), s'(\mathcal{T}_r))$$

A transition  $\delta(q, s) = (q', s', R)$  to the right is embedded in a similar fashion:

$$\begin{aligned} l_0 := q(\mathcal{T}_l, s(1), \mathcal{S}_x(\mathcal{T}_r)) &\Rightarrow l_0 := q'(s'(\mathcal{T}_l), \mathcal{S}_x(1), \mathcal{T}_r) \\ l_0 := q(\mathcal{T}_l, s(1), 0) &\Rightarrow l_0 := q'(s'(\mathcal{T}_l), \square(1), 0) \end{aligned}$$

Besides the rewriting rules induced by the Turing machine's transition function, we add rewriting rules to ensure that the embedding of any halting state can be rewritten to the distinguished halting instruction. The Turing machine halts if the transition function is not defined on the pair of the state and the current symbol. Therefore, for each such pair  $(q, s) \notin \text{dom}(\delta)$ , we add the rewrite rule

$$l_0 := q(\mathcal{T}_l, s(1), \mathcal{T}_r) \Rightarrow l_0 := \text{halted}()$$

Now that we have seen the embedding of Turing-machine configurations and transitions into instructions and rewriting rules, we show that there is nearly a bisimulation between transitions in the Turing machine and rewrites on the machine instructions. The only exception, of course, is that there may be an extra step of rewriting to reach the distinguished halting instruction.

**Theorem 1.** *For Turing-machine configurations  $c_1$  and  $c_2$ :*

1. *If  $c_1 \rightarrow c_2$  and  $i_1 = \phi(c_1)$ , then there exists an  $i_2$  such that  $i_1 \Rightarrow i_2$  and  $i_2 = \phi(c_2)$ .*
2. *If  $i_1 = \phi(c_1)$  and  $i_1 \Rightarrow i_2$ , then either there exists a  $c_2$  such that  $c_1 \rightarrow c_2$  and  $i_2 = \phi(c_2)$  or  $i_2$  is the distinguished halting instruction and  $c_1$  is a halting state.*

---

<sup>1</sup>The use of a metavariable that stands for an operator is standard practice in term rewriting, although I did not need this practice with the algebraic laws used in my algorithm. Adding this type of metavariable is a trivial extension.

*Proof.* We start with the first condition: if  $c_1 \rightarrow c_2$  and  $i_1 = \phi(c_1)$ , then there exists an  $i_2$  such that  $i_1 \Rightarrow i_2$  and  $i_2 = \phi(c_2)$ .

1. Choose a configuration  $c_1 = \langle q, (T'_l, s, T'_r) \rangle$ . We can divide the possible transitions of the Turing machine into four cases:

- (a) Case 1: The transition is  $\delta(q, s) = (q', s', L)$  for some  $q'$  and  $s'$ , and the left-hand stack is not empty:  $T'_l = S'_x T''_l$  for some symbol  $S'_x$  and some stack  $T''_l$ . The configuration  $c_2$  after the transition is  $\langle q', (T''_l, S'_x, s' T'_r) \rangle$ .

The embedding  $\phi(c_1)$  produces the instruction  $i_1$ :

$$l_0 := q(S'_x(\phi_T(T''_l)), s(1), \phi_T(T'_r)).$$

And by construction, the following rewrite rule is produced by the embedding of the Turing-machine transition:

$$l_0 := q(\mathcal{S}_x(\mathcal{T}_l), s(1), \mathcal{T}_r) \Rightarrow l_0 := q'(\mathcal{T}_l, \mathcal{S}_x(1), s'(\mathcal{T}_r))$$

If we constrain the metavariables in the rewrite rule using  $\mathcal{S}_x = S'_x$ ,  $\mathcal{T}_l = \phi_T(T''_l)$ ,  $\mathcal{T}_r = \phi_T(T'_r)$ , and substitute into the rewrite rule, we get:

$$\begin{aligned} l_0 &:= q(S'_x(\phi_T(T''_l)), s(1), \phi_T(T'_r)) \Rightarrow \\ l_0 &:= q'(\phi_T(T''_l), S'_x(1), s'(\phi_T(T'_r))) \end{aligned}$$

Using this rewrite rule, we can rewrite  $i_1$  to a new instruction  $i_2$ :

$$l_0 := q'(\phi_T(T''_l), S'_x(1), s'(\phi_T(T'_r))),$$

which is the embedding  $\phi(c_2)$ .

- (b) Case 2: The transition is  $\delta(q, s) = (q', s', L)$  for some  $q'$  and  $s'$ , and the left-hand stack is empty  $T'_l = \circ$ . The configuration  $c_2$  after the transition is  $\langle q', (\circ, \square, s' T'_r) \rangle$ . The embedding  $\phi(c_1)$  produces the instruction  $i_1$ :

$$l_0 := q(0, s(1), \phi_T(T'_r)).$$

And by construction, the following rewrite rule is produced by the embedding of the Turing-machine transition:

$$l_0 := q(0, s(1), \mathcal{T}_r) \Rightarrow l_0 := q'(0, \square(1), s'(\mathcal{T}_r))$$

If we constrain the metavariable  $\mathcal{T}_r$  in the rewrite rule using  $\mathcal{T}_r = \phi_T(T'_r)$ , and substitute into the rewrite rule, we get:

$$l_0 := q(0, s(1), \phi_T(T'_r)) \Rightarrow l_0 := q'(0, \square(1), s'(\phi_T(T'_r)))$$

Using this rewrite rule, we can rewrite  $i_1$  to a new instruction  $i_2$ :

$$l_0 := q'(0, \square(1), s'(\phi_T(T'_r))),$$

which is the embedding  $\phi(c_2)$ .

- (c) Case 3: The transition is  $\delta(q, s) = (q', s', R)$  for some  $q'$  and  $s'$ , and the right-hand stack is not empty:  $T'_r = S'_x T''_r$  for some symbol  $S'_x$  and some stack  $T''_r$ . The configuration  $c_2$  after the transition is  $\langle q', (s' T'_l, S'_x, T''_r) \rangle$ .

The embedding  $\phi(c_1)$  produces the instruction  $i_1$ :

$$l_0 := q(\phi_T(T'_l), s(1), S'_x(\phi_T(T''_r)))$$

And by construction, the following rewrite rule is produced by the embedding of the Turing-machine transition:

$$l_0 := q(\mathcal{T}_l, s(1), \mathcal{S}_x(\mathcal{T}_r)) \Rightarrow l_0 := q'(s'(\mathcal{T}_l), \mathcal{S}_x(1), \mathcal{T}_r)$$

If we constrain the metavariables in the rewrite rule using

$$\mathcal{S}_x = S'_x, \mathcal{T}_l = \phi_T(T'_l), \text{ and } \mathcal{T}_r = \phi_T(T''_r),$$

then substitute into the rewrite rule, we get:

$$l_0 := q(\phi_T(T'_l), s(1), S'_x(\phi_T(T''_r))) \Rightarrow$$

$$l_0 := q'(s'(\phi_T(T'_l)), S'_x(1), \phi_T(T''_r))$$

Using this rewrite rule, we can rewrite  $i_1$  to a new instruction  $i_2$ :

$$l_0 := q'(s'(\phi_T(T'_l)), S'_x(1), \phi_T(T''_r)),$$

which is the embedding  $\phi(c_2)$ .

- (d) Case 4: The transition is  $\delta(q, s) = (q', s', R)$  for some  $q'$  and  $s'$ , and the right-hand stack is empty  $T'_r = \circ$ . The configuration  $c_2$  after the transition is  $\langle q', (s' T'_l, \square, \circ) \rangle$ . The embedding  $\phi(c_1)$  produces the instruction  $i_1$ :

$$l_0 := q(\phi_T(T'_l), s(1), 0)$$

And by construction, the following rewrite rule is produced by the embedding of the Turing-machine transition:

$$l_0 := q(\mathcal{T}_l, s(1), 0) \Rightarrow l_0 := q'(s'(\mathcal{T}_l), \square(1), 0)$$

If we constrain the metavariables  $\mathcal{T}_l$  in the rewrite rule using  $\mathcal{T}_l = \phi_T(T'_l)$ , and substitute into the rewrite rule, we get:

$$l_0 := q(\phi_T(T'_l), s(1), 0) \Rightarrow l_0 := q'(s'(\phi_T(T'_l)), \square(1), 0)$$

Using this rewrite rule, we can rewrite  $i_1$  to a new instruction  $i_2$ :

$$l_0 := q'(s'(\phi_T(T'_l)), \square(1), 0),$$

which is the embedding  $\phi(c_2)$ .

2. Now, we prove the converse: if  $i_1 = \phi(c_1)$  and  $i_1 \Rightarrow i_2$ , then there exists a  $c_2$  such that  $c_1 \rightarrow c_2$  and  $i_2 = \phi(c_2)$ .

Again, choose a configuration  $c_1 = \langle q, (T'_l, s, T'_r) \rangle$ . The embedding  $\phi(c_1)$  produces the instruction  $i_1$ :

$$l_0 := q(\phi_T(T'_l), s(1), \phi_T(T'_r))$$

We divide the possible rewrite on  $i_1$  into five cases. The first four cases cover the rewrite rules that are produced by the embedding of the Turing machine's transition function. The fifth case covers the rewrite rule introduced to reach the distinguished halting instruction.

- (a) Case 1: The form of the rewrite rule is:

$$l_0 := q(\mathcal{S}_x(\mathcal{T}_l), s(1), \mathcal{T}_r) \Rightarrow l_0 := q'(\mathcal{T}_l, \mathcal{S}_x(1), s'(\mathcal{T}_r))$$

For the rewrite rule to apply to  $i_1$ , it must be the case that  $\phi_T(T'_l) = \mathcal{S}'_x(\phi_T(T''_l))$  for some operator  $\mathcal{S}'_x$  and some stack  $T''_l$ . Therefore, we can state more precisely that  $i_1$  is

$$l_0 := q(\mathcal{S}'_x(\phi_T(T''_l)), s(1), \phi_T(T'_r))$$

Using  $\phi_T(T'_l) = \mathcal{S}'_x(\phi_T(T''_l))$  and the definition of  $\phi_T$ , we can conclude that  $T'_l = \mathcal{S}'_x T''_l$ . Therefore, we can state more precisely that  $c_1$  is  $\langle q, (\mathcal{S}'_x T''_l, s, T'_r) \rangle$ .

To apply the rewrite rule, we must constrain the metavariables using

$$\mathcal{S}_x = \mathcal{S}'_x, \mathcal{T}_l = \phi_T(T''_l), \text{ and } \mathcal{T}_r = \phi_T(T'_r),$$

then substitute into the rewrite rule to get:

$$l_0 := q(\mathcal{S}'_x(\phi_T(T''_l)), s(1), \phi_T(T'_r)) \Rightarrow$$

$$l_0 := q'(\phi_T(T_l''), S'_x(1), s'(\phi_T(T_r')))$$

Using this rewrite rule,  $i_1$  is rewritten to  $i_2$ :

$$l_0 := q'(\phi_T(T_l''), S'_x(1), s'(\phi_T(T_r')))$$

By construction, this form of rewriting rule can only be introduced if the Turing machine has the transition  $\delta(q, s) = (q', s', L)$ . Applying this transition to the configuration  $c_1$ , we get the configuration  $c_2$ :  $\langle q', (T_l'', S'_x, s'T_r') \rangle$ , and we can conclude that  $i_2 = \phi(c_2)$ .

(b) Case 2: The form of the rewrite rule is:

$$l_0 := q(0, s(1), \mathcal{T}_r) \Rightarrow l_0 := q'(0, \square(1), s'(\mathcal{T}_r))$$

For the rewrite rule to apply to  $i_1$ , it must be the case that  $\phi_T(T_l') = 0$ . Therefore, we can state more precisely that  $i_1$  is

$$l_0 := q(0, s(1), \phi_T(T_r'))$$

Using  $\phi_T(T_l') = 0$  and the definition of  $\phi_T$ , we can also conclude that  $T_l' = \circ$ . Therefore, we can state more precisely that  $c_1$  is  $\langle q, (\circ, s, T_r') \rangle$ .

To apply the rewrite rule, we must constrain the metavariable  $\mathcal{T}_r = \phi_T(T_r')$ , then substitute into the rewrite rule to get:

$$l_0 := q(0, s(1), \phi_T(T_r')) \Rightarrow l_0 := q'(0, \square(1), s'(\phi_T(T_r')))$$

Using this rewrite rule,  $i_1$  is rewritten to  $i_2$ :

$$l_0 := q'(0, \square(1), s'(\phi_T(T_r')))$$

By construction, this form of rewriting rule can only be introduced if the Turing machine has the transition  $\delta(q, s) = (q', s', L)$ . Applying this transition to the configuration  $c_1$ , we get the configuration  $c_2$ :  $\langle q', (\circ, \square, s'T_r') \rangle$ , and we can conclude that  $i_2 = \phi(c_2)$ .

(c) Case 3: The form of the rewrite rule is:

$$l_0 := q(\mathcal{T}_l, s(1), \mathcal{S}_x(\mathcal{T}_r)) \Rightarrow l_0 := q'(s'(\mathcal{T}_l), \mathcal{S}_x(1), \mathcal{T}_r)$$

For the rewrite rule to apply to  $i_1$ , it must be the case that  $\phi_T(T_r') = S'_x(\phi_T(T_r''))$  for some operator  $S'_x$  and some stack  $T_r''$ . Therefore, we can state more precisely that  $i_1$  is

$$l_0 := q(\phi_T(T_l'), s(1), S'_x(\phi_T(T_r'')))$$

Using  $\phi_T(T'_r) = S'_x(\phi_T(T''_r))$  and the definition of  $\phi_T$ , we can conclude that  $T'_r = S'_x T''_r$ . Therefore, we can state more precisely that  $c_1$  is  $\langle q, (T'_l, s, S'_x T''_r) \rangle$ .

To apply the rewrite rule, we must constrain the metavariables using

$$S_x = S'_x, T_l = \phi_T(T'_l), \text{ and } T_r = \phi_T(T''_r),$$

then substitute into the rewrite rule to get:

$$\begin{aligned} l_0 &:= q(\phi_T(T'_l), s(1), S'_x(\phi_T(T''_r))) \Rightarrow \\ l_0 &:= q'(s'(\phi_T(T'_l)), S'_x(1), \phi_T(T''_r)) \end{aligned}$$

Using this rewrite rule,  $i_1$  is rewritten to  $i_2$ :

$$l_0 := q'(s'(\phi_T(T'_l)), S'_x(1), \phi_T(T''_r))$$

By construction, this form of rewriting rule can only be introduced if the Turing machine has the transition  $\delta(q, s) = (q', s', R)$ . Applying this transition to the configuration  $c_1$ , we get the configuration  $c_2: \langle q', (s'T'_l, S'_x, T''_r) \rangle$ , and we can conclude that  $i_2 = \phi(c_2)$ .

(d) Case 4: The form of the rewrite rule is:

$$l_0 := q(T_l, s(1), 0) \Rightarrow l_0 := q'(s'(T_l), \square(1), 0)$$

For the rewrite rule to apply to  $i_1$ , it must be the case that  $\phi_T(T'_r) = 0$ . Therefore, we can state more precisely that  $i_1$  is

$$l_0 := q(\phi_T(T'_l), s(1), 0)$$

Using  $\phi_T(T'_r) = 0$  and the definition of  $\phi_T$ , we can also conclude that  $T'_r = \circ$ . Therefore, we can state more precisely that  $c_1$  is  $\langle q, (T'_l, s, \circ) \rangle$ .

To apply the rewrite rule, we must constrain the metavariable  $T_l = \phi_T(T'_l)$ , then substitute into the rewrite rule to get:

$$l_0 := q(\phi_T(T'_l), s(1), 0) \Rightarrow l_0 := q'(s'(\phi_T(T'_l)), \square(1), 0)$$

Using this rewrite rule,  $i_1$  is rewritten to  $i_2$ :

$$l_0 := q'(s'(\phi_T(T'_l)), \square(1), 0)$$

By construction, this form of rewriting rule can only be introduced if the Turing machine has the transition  $\delta(q, s) = (q', s', R)$ . Applying this transition to the configuration  $c_1$ , we get the configuration  $c_2: \langle q', (s'T'_l, \square, \circ) \rangle$ , and we can conclude that  $i_2 = \phi(c_2)$ .

(e) Case 5: The form of the rewrite rule is:

$$l_0 := q(\mathcal{T}_l, s(1), \mathcal{T}_r) \Rightarrow l_0 := \text{halted}()$$

This rewrite results in the instruction  $i_2 = l_0 := \text{halted}()$ , which is the distinguished halting instruction.

And by construction, this rule is introduced only if the machine has no transition on the state  $q$  with current symbol  $s$ , (i.e.  $(q, s) \notin \text{dom}(\delta)$ ). Therefore, the configuration  $c_1$  must be in a halting state.

□

Now, we can state the main theorem, which shows that an algorithm that decides the Instruction Equivalence Problem can be used to decide the halting problem: given the initial configuration  $c$  of a Turing machine, we can use the algorithm to decide whether  $\phi(c)$  is equivalent to the distinguished halting instruction. If so, then the Turing machine will halt; otherwise, it will not. Therefore, because the halting problem is undecidable, the Instruction Equivalence Problem must also be undecidable.

**Theorem 2.** *For Turing-machine configurations  $c_1$  and  $c_n$ , as well as the distinguished halting instruction  $i_h$ :*

1. *If  $c_1 \rightarrow^* c_n$  and  $c_n$  is a halting state, then  $\phi(c_1) \Rightarrow^* i_h$ .*
2. *If  $\phi(c_1) \Rightarrow^* i_h$ , then  $c_1 \rightarrow^* c_n$  where  $c_n$  is a halting state.*

*Proof.* Given Theorem 1, the proof is fairly straightforward:

1. First, we show that if  $c_1 \rightarrow^* c_n$  and  $c_n$  is a halting state, then  $\phi(c_1) \Rightarrow^* i_h$ .  
The proof proceeds by induction on the length of the relation  $c_1 \rightarrow^* c_n$ :

- *Base case:* 0 steps, so  $c_1 = c_n$ . Choose  $c_1 = \langle q, (\mathcal{T}_l, s, \mathcal{T}_r) \rangle$ . Therefore,  $\phi(c_1) = l_0 := q(\phi_T(\mathcal{T}_l), s(1), \phi_T(\mathcal{T}_r))$ . And because  $c_1$  is a halting state, the rewrite rules include

$$l_0 := q(\mathcal{T}_l, s(1), \mathcal{T}_r) \Rightarrow l_0 := \text{halted}()$$

We can apply this rewrite rule to show  $\phi(c_1) \Rightarrow l_0 := \text{halted}()$ .

- *Inductive case:*  $c_1 \rightarrow c_2 \rightarrow^{n-1} c_n$ . The inductive hypothesis states that the theorem holds for  $n - 1$  steps: if  $c_2 \rightarrow^{n-1} c_n$  and  $c_n$  is a halting state, then  $\phi(c_2) \Rightarrow^* i_h$ . Given  $c_1 \rightarrow c_2$ , we invoke Theorem 1 on page 149 to show that  $\phi(c_1) \Rightarrow \phi(c_2)$ . Therefore, we can conclude that  $\phi(c_1) \Rightarrow^* i_h$ .

2. Now, we show that if  $\phi(c_1) \Rightarrow^* i_h$ , then  $c_1 \rightarrow^* c_n$  where  $c_n$  is a halting state. The proof proceeds by induction on the length of the relation  $\phi(c_1) \Rightarrow^* i_h$ :

- *Base case:* There is no configuration  $c_1$  such that  $\phi(c_1) = i_h$  (given our definition of  $\phi$ ), so the non-trivial base case is one step.

Choose  $\phi(c_1) = l_0 := q(\phi_T(T_l), s(0), \phi_T(T_r))$ .

Using the definition of *phi*,  $c_1 = \langle q, (T_l, s, T_r) \rangle$ . By construction, the only rewrite rule that can produce  $i_h$  is

$$l_0 := q(\mathcal{T}_l, s(1), \mathcal{T}_r) \Rightarrow l_0 := \text{halted}()$$

But this rule is added only if  $(q, s) \notin \delta$ . Therefore,  $c_1$  is a halting state, and of course  $c_1 \rightarrow^* c_1$ .

- *Inductive case:*  $i_1 \Rightarrow i_2 \Rightarrow^{n-1} i_h$ , where  $i_1 = \phi(c_1)$ . The inductive hypothesis states that the theorem holds for  $n - 1$  steps: if  $i_2 \Rightarrow^{n-1} i_h$  then  $c_2 \rightarrow^* c_n$ , where  $i_2 = \phi(c_2)$  and  $c_n$  is a halting state. Given  $i_1 \Rightarrow i_2$ , we invoke Theorem 1 on page 149, which says that one of two conditions hold:

- There exists a  $c_2$  such that  $c_1 \rightarrow c_2$  and  $i_2 = \phi(c_2)$ . In this case, we can use the inductive hypothesis to show that  $c_1 \rightarrow c_2 \rightarrow^* c_n$ , where  $c_n$  is a halting state.
- Otherwise,  $i_2$  is the distinguished halting instruction and  $c_1$  is a halting state. And of course,  $c_1 \Rightarrow^* c_1$ .

□

# Appendix B

## RTL operators

The following tables contains the RTL operators used by the Quick C-- compiler. The names and types of the operators are declared in the compiler, which, among other uses, allows us to typecheck the RTLs. In addition to the collection of standard arithmetic and logical operators, we include useful constants, such as *true*.

Operator	Type	Description
NaN	$\forall n, m : \#n \text{ bits} \rightarrow \#m \text{ bits}$	floating point not a number
add	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	addition
add_overflows	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	overflow of addition
addc	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#1 \text{ bits} \rightarrow \#n \text{ bits}$	add with carry
and	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	bitwise and
bit	$\text{bool} \rightarrow \#1 \text{ bits}$	convert bool to 1 bit
bitExtract	$\forall n, m : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#m \text{ bits}$	extract part of a bit vector
bitInsert	$\forall n, m : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#m \text{ bits} \rightarrow \#n \text{ bits}$	insert into a bit vector
bool	$\#1 \text{ bits} \rightarrow \text{bool}$	convert 1 bit to bool
borrow	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#1 \text{ bits} \rightarrow \#1 \text{ bits}$	compute borrow of subtraction
carry	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#1 \text{ bits} \rightarrow \#1 \text{ bits}$	compute carry of addition
com	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits}$	bitwise complement

Operator	Type	Description
<code>conjoin</code>	$\text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$	logical conjunction
<code>disjoin</code>	$\text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$	logical disjunction
<code>div</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	division rounding toward negative infinity
<code>div_overflows</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	overflow of division
<code>divu</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	unsigned division
<code>eq</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	equality on bit vectors
<code>f2f</code>	$\forall n, m : \#n \text{ bits} \rightarrow \#2 \text{ bits} \rightarrow \#m \text{ bits}$	floating-point width conversion
<code>f2i</code>	$\forall n, m : \#n \text{ bits} \rightarrow \#2 \text{ bits} \rightarrow \#m \text{ bits}$	convert float to int
<code>fabs</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits}$	floating-point absolute value
<code>fadd</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#2 \text{ bits} \rightarrow \#n \text{ bits}$	floating-point addition
<code>false</code>	<code>bool</code>	Boolean false
<code>fcmp</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#2 \text{ bits}$	floating-point comparison
<code>fdiv</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#2 \text{ bits} \rightarrow \#n \text{ bits}$	floating-point division
<code>feq</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare floats for equality
<code>fge</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare floats for greater-than or equal
<code>fgt</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare floats for greater-than
<code>fle</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare floats for less-than or equal
<code>float_eq</code>	$\#2 \text{ bits}$	value indicating floating-point equal
<code>float_gt</code>	$\#2 \text{ bits}$	value indicating floating-point greater-than
<code>float_lt</code>	$\#2 \text{ bits}$	value indicating floating-point less-than
<code>flt</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare floats for less-than or
<code>fmul</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#2 \text{ bits} \rightarrow \#n \text{ bits}$	floating-point multiply
<code>fmulx</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow 2 * \#n \text{ bits}$	floating-point multiply-extended
<code>fne</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare floats for inequality
<code>fneg</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits}$	floating-point negation
<code>forordered</code>	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare floats for ordered
<code>fsqrt</code>	$\forall n : \#n \text{ bits} \rightarrow \#2 \text{ bits} \rightarrow \#n \text{ bits}$	floating-point square root

Operator	Type	Description
fsub	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#2 \text{ bits} \rightarrow \#n \text{ bits}$	floating-point subtraction
funordered	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare floats for unordered
ge	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare greater-than or equal
geu	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare greater-than or equal (unsigned)
gt	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare greater-than
gtu	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare greater-than (unsigned)
gx	$\forall n, m : \#n \text{ bits} \rightarrow \#m \text{ bits}$	extension with arbitrary (garbage) bits added
i2f	$\forall n, m : \#n \text{ bits} \rightarrow \#2 \text{ bits} \rightarrow \#m \text{ bits}$	convert int to float
le	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare less-than or equal
leu	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare less than or equal (unsigned)
lobits	$\forall n, m : \#n \text{ bits} \rightarrow \#m \text{ bits}$	extract low bits
lt	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare less-than
ltu	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare less-than (unsigned)
minf	$\forall n : \#n \text{ bits}$	negative infinity
mod	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	modulus (works with div)
modu	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	modulus (unsigned, works with div)
mul	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	multiply
mul_overflows	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	overflow of multiply
mulu_overflows	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	overflow of unsigned multiply
mulux	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow 2 * \#n \text{ bits}$	extended multiply (unsigned)
mulx	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow 2 * \#n \text{ bits}$	extended multiply
mzero	$\forall n : \#n \text{ bits}$	negative zero
ne	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	compare for inequality
neg	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits}$	integer negation
not	$\text{bool} \rightarrow \text{bool}$	Boolean not
or	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	bitwise or
pinf	$\forall n : \#n \text{ bits}$	positive infinity
popcnt	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits}$	population count
pzero	$\forall n : \#n \text{ bits}$	positive zero
quot	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	division (round toward 0)

Operator	Type	Description
quot_overflows	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	overflow of quot
rem	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	remainder (works with quot)
rotr	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	rotate right
rotl	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	rotate left
round_down	$\#2 \text{ bits}$	floating-point round-down
round_nearest	$\#2 \text{ bits}$	floating-point round-to-nearest
round_up	$\#2 \text{ bits}$	floating-point round-up
round_zero	$\#2 \text{ bits}$	floating-point round-towards-zero
shl	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	shift left
shra	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	shift right (arithmetic)
shrl	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	shift right (logical)
sub	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	subtract
sub_overflows	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \text{bool}$	overflow of subtract
subb	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#1 \text{ bits} \rightarrow \#n \text{ bits}$	subtract with borrow
sx	$\forall n, m : \#n \text{ bits} \rightarrow \#m \text{ bits}$	sign extend
true	bool	Boolean true
unordered	$\#2 \text{ bits}$	floating-point unordered value
xor	$\forall n : \#n \text{ bits} \rightarrow \#n \text{ bits} \rightarrow \#n \text{ bits}$	exclusive-or
zx	$\forall n, m : \#n \text{ bits} \rightarrow \#m \text{ bits}$	zero extend

# Appendix C

## Algebraic laws

This appendix lists the algebraic laws used in my implementation (Section 6.1.1 on page 82). These laws do not represent a complete algebraic theory: they represent only a small subset of laws we have needed to develop our back ends. I have organized the algebraic laws in groups, according to how they function, but in practice the groups are ignored. For the most part, I have elided widths, making the laws significantly easier to read.

<code>add(x, 0)</code>	<code>=</code>	<code>x</code>
<code>add(0, x)</code>	<code>=</code>	<code>x</code>
<code>or(x, 0)</code>	<code>=</code>	<code>x</code>
<code>or(0, x)</code>	<code>=</code>	<code>x</code>
<code>shl(x, 0)</code>	<code>=</code>	<code>x</code>
<code>shra(x, 0)</code>	<code>=</code>	<code>x</code>
<code>shrl(x, 0)</code>	<code>=</code>	<code>x</code>
<code>sub(x, 0)</code>	<code>=</code>	<code>x</code>

Table C.1: Operator identities.

<code>com(com(x))</code>	<code>=</code>	<code>x</code>
<code>neg(neg(x))</code>	<code>=</code>	<code>x</code>
<code>bitExtract(0, zx(x))</code>	<code>=</code>	<code>x</code>
<code>bitExtract(0, gx(x))</code>	<code>=</code>	<code>x</code>

Table C.2: Operator inverses.

<code>quotx(sx(x), x')</code>	<code>=</code>	<code>quot(x, x')</code>
<code>remx(sx(x), x')</code>	<code>=</code>	<code>rem(x, x')</code>
<code>divux(zx(x), x')</code>	<code>=</code>	<code>divu(x, x')</code>
<code>modux(zx(x), x')</code>	<code>=</code>	<code>modu(x, x')</code>
<code>bitExtract(0, mulx(x, x'))</code>	<code>=</code>	<code>mul(x, x')</code>
<code>bitExtract(0, mulux(x, x'))</code>	<code>=</code>	<code>mulu(x, x')</code>
<code>sub(x, mul(quot(x, x'), x'))</code>	<code>=</code>	<code>rem(x, x')</code>
<code>bitExtract(0, mulx(x, x'))</code>	<code>=</code>	<code>mulu(x, x')</code>
<code>sub(x, mulu(divu(x, x'), x'))</code>	<code>=</code>	<code>modu(x, x')</code>
<code>or(shl(zx(0), x'), zx(x))</code>	<code>=</code>	<code>zx(x)</code>
<code>shrl(shl(gx(x), zx(bitExtract(0, sub(x', x'')))), zx(bitExtract(0, sub(x', x''))))</code>	<code>=</code>	<code>zx(x)</code>
<code>shra(shl(gx(x), zx(bitExtract(0, sub(x', x'')))), zx(bitExtract(0, sub(x', x''))))</code>	<code>=</code>	<code>sx(x)</code>
<code>shra(shl(zx(x), zx(bitExtract(0, sub(x', x'')))), zx(bitExtract(0, sub(x', x''))))</code>	<code>=</code>	<code>sx(x)</code>
<code>or(shl(zx(shra(x, zx(bitExtract(0, sub(w, 1))))), w), zx(x:w bits):2w bits)</code>	<code>=</code>	<code>sx(x)</code>
<code>(if x = minint then</code>		<code>if div_overflows(x, y)</code>
<code>  (if y = -1 then goto L<sub>T</sub> else goto L<sub>F</sub>)</code>	<code>=</code>	<code>then goto L<sub>T</sub></code>
<code>  else goto L<sub>F</sub>)</code>		<code>else goto L<sub>F</sub></code>

Table C.3: Operator implementations.

$\text{shrl}(\text{shl}(x, \text{zx}(\text{bitExtract}(0, \text{sub}(x', x'')))),$	$=$	$\text{zx}(\text{bitExtract}(0, x))$
$\text{zx}(\text{bitExtract}(0, \text{sub}(x', x''))))$		
$\text{shra}(\text{shl}(x, \text{zx}(\text{bitExtract}(0, \text{sub}(x', 1))))),$	$=$	$\text{sx}(\text{bitExtract}(0, x))$
$\text{zx}(\text{bitExtract}(0, \text{sub}(x', 1))))$		
$\text{or}(\text{zx}(\text{bitExtract}(0, x)), \text{shl}(\text{shrl}(x, x'), x'))$	$=$	$x$
$\text{or}(\text{shl}(\text{shrl}(x, x'), x'), \text{zx}(\text{bitExtract}(0, x)))$	$=$	$x$
$\text{add}(\text{shl}(\text{sx}(\text{add}(\text{shrl}(\text{bitExtract}(0, x), 15),$	$=$	$x$
$\text{bitExtract}(16, x))), 16), \text{sx}(\text{bitExtract}(0, x)))$		
$\text{bitExtract}(0, \text{zx}(x))$	$=$	$x$
$\text{sx}(\text{bitExtract}(x, 0))$	$=$	$0$
$\text{zx}(\text{bitExtract}(x, 0))$	$=$	$0$
$\text{bitExtract}(0, x@0)$	$=$	$\text{bitExtract}(0, x)$
$\text{gx}(\text{gx}(x))$	$=$	$\text{gx}(x)$
$\text{bitExtract}(x', x)$	$=$	$x@x'$
$\text{lobits}(x)$	$=$	$x@0$
$\text{bitExtract}(\text{zx}(\text{bitExtract}(0, 0)), \text{zx}(x))$	$=$	$x$
$\text{bitExtract}(\text{zx}(\text{bitExtract}(0, 0)), \text{gx}(x))$	$=$	$x$
$\text{com}(\text{bitExtract}(0, \text{com}(x)))$	$=$	$\text{bitExtract}(0, x)$
$\text{shl}(\text{zx}(\text{bitExtract}(0, x)), x')$	$=$	$\text{shl}(x, x')$
$\text{shl}(\text{zx}(\text{lobits}(\text{shrl}(x, 32))), 32)$	$=$	$\text{shl}(\text{shrl}(x, 32), 32)$
$\text{i2f}(\text{sx}(x), x')$	$=$	$\text{i2f}(x, x')$
$\text{bitExtract}(0, \text{mulx}(x, x'))$	$=$	$\text{bitExtract}(0, \text{mulux}(x, x'))$
$x@32 := \text{bitExtract}(0, x');$		
$x@0 := \text{bitExtract}(0, \text{shrl}(x', 32))$	$=$	$x := x'$
		on big-endian machine
$x@0 := \text{bitExtract}(0, x');$		
$x@32 := \text{bitExtract}(0, \text{shrl}(x', 32))$	$=$	$x := x'$
		on little-endian machine
$x''@32 := \text{xor}(x', \text{shl}(32768, 16));$		
$x''@0 := \text{shl}(17200, \text{zx}(\text{bitExtract}(0, 16)));$		
$x := \text{fsub}(x'', \text{or}(\text{shl}(\text{zx}(\text{shl}(17200,$		
$\text{zx}(\text{bitExtract}(0, 16))))), 32),$		
$\text{zx}(\text{shl}(32768, 16))), c[3]@30)$	$=$	$x := \text{i2f}(x', (c[3]@30)$
		on big-endian machine
$x@32 := x';$		
$x@0 := \text{shra}(x', \text{zx}(\text{bitExtract}(0, \text{sub}(32, 1))))$	$=$	$x := \text{sx}(x')$
		on big-endian machine
$x@0 := x';$		
$x@32 := \text{shra}(x', \text{zx}(\text{bitExtract}(0, \text{sub}(32, 1))))$	$=$	$x := \text{sx}(x')$
		on little-endian machine

Table C.4: Aggregating identities, along with some leftovers that don't properly fit into the previous categories.

<code>x86_ah2flags(x86_flags2ah(x))</code>	<code>= x</code>
<code>x86_carrybit(x86_setcarry(x', x))</code>	<code>= x</code>
<code>x86_flags2ah(x86_sbbflags(x, x', x''))</code>	<code>= gx(borrow(x, x', x''))</code>
<code>x86_flags2ah(x86_adcflags(x, x', x''))</code>	<code>= gx(carry(x, x', x''))</code>
<code>x86_z(x86_subflags(x, x'))</code>	<code>= eq(x, x')</code>
<code>x86_nbe(x86_ah2flags(bitExtract(8, gx(x86_fcmp(x', x))))))</code>	<code>= flt(x, x')</code>
<code>x86_nb(x86_ah2flags(bitExtract(8, gx(x86_fcmp(x', x))))))</code>	<code>= fle(x, x')</code>
<code>x86_nbe(x86_ah2flags(bitExtract(8, gx(x86_fcmp(x', x))))))</code>	<code>= fgt(x', x)</code>
<code>x86_nb(x86_ah2flags(bitExtract(8, gx(x86_fcmp(x', x))))))</code>	<code>= fge(x', x)</code>
<code>x86_np(x86_ah2flags(bitExtract(8, gx(x86_fcmp(x', x))))))</code>	<code>= fordered(x', x)</code>
<code>x86_p(x86_ah2flags(bitExtract(8, gx(x86_fcmp(x', x))))))</code>	<code>= funordered(x', x)</code>
<code>x86_z(x86_ah2flags(bitExtract(8, xor(64, and(69, gx(x86_fcmp(x', x))))))</code>	<code>= feq(x', x)</code>
<code>x86_nz(x86_ah2flags(bitExtract(8, xor(64, and(69, gx(x86_fcmp(x', x))))))</code>	<code>= fne(x', x)</code>
<code>x86_z(x86_subflags(x, x'))</code>	<code>= eq(x, x')</code>
<code>x86_nz(x86_subflags(x, x'))</code>	<code>= ne(x, x')</code>
<code>x86_l(x86_subflags(x, x'))</code>	<code>= lt(x, x')</code>
<code>x86_le(x86_subflags(x, x'))</code>	<code>= le(x, x')</code>
<code>x86_nle(x86_subflags(x, x'))</code>	<code>= gt(x, x')</code>
<code>x86_nl(x86_subflags(x, x'))</code>	<code>= ge(x, x')</code>
<code>x86_b(x86_subflags(x, x'))</code>	<code>= ltu(x, x')</code>
<code>x86_be(x86_subflags(x, x'))</code>	<code>= leu(x, x')</code>
<code>x86_nbe(x86_subflags(x, x'))</code>	<code>= gtu(x, x')</code>
<code>x86_nb(x86_subflags(x, x'))</code>	<code>= geu(x, x')</code>
<code>x86_o(x86_subflags(x, x'))</code>	<code>= sub_overflows(x, x')</code>
<code>x86_o(x86_addflags(x, x'))</code>	<code>= add_overflows(x, x')</code>
<code>x86_o(x86_mulflags(x, x'))</code>	<code>= mul_overflows(x, x')</code>
<code>x86_o(x86_mluxflags(x, x'))</code>	<code>= mulu_overflows(x, x')</code>

Table C.5: Machine-specific laws for the x86.

<code>ppc_carrybit(ppc_setcarry(x))</code>	<code>= x</code>
<code>ppc_shracarry(or(neg(2), x), zx(1))</code>	<code>= bitExtract(0, x)</code>
<code>bitExtract(0, ppc_FPFlags(c[3]))</code>	<code>= c[3]</code>
<code>ppc_eq(ppc_signed_cmp(x, x'))</code>	<code>= eq(x, x')</code>
<code>ppc_ne(ppc_signed_cmp(x, x'))</code>	<code>= ne(x, x')</code>
<code>ppc_lt(ppc_signed_cmp(x, x'))</code>	<code>= lt(x, x')</code>
<code>ppc_le(ppc_signed_cmp(x, x'))</code>	<code>= le(x, x')</code>
<code>ppc_gt(ppc_signed_cmp(x, x'))</code>	<code>= gt(x, x')</code>
<code>ppc_ge(ppc_signed_cmp(x, x'))</code>	<code>= ge(x, x')</code>
<code>ppc_lt(ppc_unsigned_cmp(x, x'))</code>	<code>= ltu(x, x')</code>
<code>ppc_le(ppc_unsigned_cmp(x, x'))</code>	<code>= leu(x, x')</code>
<code>ppc_gt(ppc_unsigned_cmp(x, x'))</code>	<code>= gtu(x, x')</code>
<code>ppc_ge(ppc_unsigned_cmp(x, x'))</code>	<code>= geu(x, x')</code>
<code>ppc_eq(ppc_fpcmp(x, x'))</code>	<code>= feq(x, x')</code>
<code>ppc_ne(ppc_fpcmp(x, x'))</code>	<code>= fne(x, x')</code>
<code>ppc_lt(ppc_fpcmp(x, x'))</code>	<code>= flt(x, x')</code>
<code>ppc_le(ppc_fpcmp(x, x'))</code>	<code>= fle(x, x')</code>
<code>ppc_gt(ppc_fpcmp(x, x'))</code>	<code>= fgt(x, x')</code>
<code>ppc_ge(ppc_fpcmp(x, x'))</code>	<code>= fge(x, x')</code>
<code>ppc_ov(ppc_fpcmp(x, x'))</code>	<code>= fordered(x, x')</code>
<code>ppc_nov(ppc_fpcmp(x, x'))</code>	<code>= funordered(x, x')</code>
<code>ppc_ov(ppc_addcmpovflags(x, x', ppc_getSticky(x'', x''')))</code>	<code>= add_overflows(x, x')</code>
<code>ppc_ov(ppc_subcmpovflags(x, x', ppc_getSticky(x'', x''')))</code>	<code>= sub_overflows(x, x')</code>
<code>ppc_ov(ppc_mulcmpovflags(x, x', ppc_getSticky(x'', x''')))</code>	<code>= mul_overflows(x, x')</code>
<code>shrl(ppc_adccmpovflags(x, x', x'', x'''), zx(bitExtract(0, 29)))</code>	<code>= gx(carry(x, x', x''))</code>
<code>shrl(com(ppc_sbbcmpovflags(x, x', x'', x'''), zx(bitExtract(0, 29)))</code>	<code>= gx(borrow(x, x', x''))</code>
<code>ppc_extractFPFlags(ppc_FPFlags(x), 255)</code>	<code>= x</code>
<code>x := ppc_FPFlags(c[3])</code>	<code>= x@32:#1 bits:= bitExtract(0, ppc_FPFlags(c[3]))</code>
<code>x''' := ppc_i_ext(f2i(x', x'')); x := x'''@32:#1 bits</code>	<code>= x := f2i(x', x'')</code>
<code>c[3] := ppc_setFPBit(ppc_setFPBit(c[3], 30, 0), 31, 0)</code>	<code>= c[3]@30:#2 bits := 0</code>
<code>c[3] := ppc_setFPBit(ppc_setFPBit(c[3], 30, 0), 31, 1)</code>	<code>= c[3]@30:#2 bits := 1</code>
<code>c[3] := ppc_setFPBit(ppc_setFPBit(c[3], 30, 1), 31, 0)</code>	<code>= c[3]@30:#2 bits := 2</code>
<code>c[3] := ppc_setFPBit(ppc_setFPBit(c[3], 30, 1), 31, 1)</code>	<code>= c[3]@30:#2 bits := 3</code>
<code>or(and(c[3], shrl(-1, 2)), shl(x, zx(bitExtract(0, 30))))</code>	<code>= bitInsert(30, c[3], x)</code>

Table C.6: Machine-specific laws for the PowerPC.

<code>arm_eq(arm_flags(sub(x, x')))</code>	<code>= eq(x, x')</code>
<code>arm_ne(arm_flags(sub(x, x')))</code>	<code>= ne(x, x')</code>
<code>arm_lt(arm_flags(sub(x, x')))</code>	<code>= lt(x, x')</code>
<code>arm_le(arm_flags(sub(x, x')))</code>	<code>= le(x, x')</code>
<code>arm_gt(arm_flags(sub(x, x')))</code>	<code>= gt(x, x')</code>
<code>arm_ge(arm_flags(sub(x, x')))</code>	<code>= ge(x, x')</code>
<code>arm_unsigned_lt(arm_flags(sub(x, x')))</code>	<code>= ltu(x, x')</code>
<code>arm_unsigned_le(arm_flags(sub(x, x')))</code>	<code>= leu(x, x')</code>
<code>arm_unsigned_gt(arm_flags(sub(x, x')))</code>	<code>= gtu(x, x')</code>
<code>arm_unsigned_ge(arm_flags(sub(x, x')))</code>	<code>= geu(x, x')</code>
<code>arm_overflow_set(arm_flags(sub(x, x')))</code>	<code>= sub_overflows(x, x')</code>
<code>arm_overflow_set(arm_flags(add(x, x')))</code>	<code>= add_overflows(x, x')</code>
<code>arm_overflow_set(arm_flags(mul(x, x')))</code>	<code>= mul_overflows(x, x')</code>
<code>arm_overflow_set(arm_flags(mulu(x, x')))</code>	<code>= mulu_overflows(x, x')</code>

Table C.7: Machine-specific laws for the ARM.

## References

- Martn Abadi, Mihai Budiu, Ivar Erlingsson, and Jay Ligatti. 2005. A theory of secure control flow. In *International Conference on Formal Engineering Methods*, pages 111–124.
- A. V. Aho, R. Sethi, and J. D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass.
- Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. 1989. Code generation using tree matching and dynamic programming. *Transactions on Programming Languages and Systems*, 11(4): 491–516.
- Andrew W. Appel. 1998. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, United Kingdom.
- Leo Bachmair, Ta Chen, and I. V. Ramakrishnan. 1993. Associative-commutative discrimination nets. In M.C. Gaudel and J.P. Jouannaud, editors, *Theory and Practice of Software Development, LNCS volume 668*, pages 61–74. Springer.
- J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. 1963. Revised report on the algorithm language ALGOL 60. *Communications of the ACM*, 6(1):1–17.
- Mark W. Bailey and Jack W. Davidson. 1995. A formal model and specification language for procedure calling conventions. In *Symposium on Principles of Programming Languages*, pages 298–310, New York, NY. ACM.
- Mark W. Bailey and Jack W. Davidson. 2003. Automatic detection and diagnosis of faults in generated code for procedure calls. *IEEE Transactions on Software Engineering*, 29(11):1031–1042.
- Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12.

- Joel F. Bartlett. 1988. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Western Research Laboratory, Digital Equipment Corporation.
- Joel F. Bartlett. 1989. SCHEME→C: A portable Scheme-to-C compiler. Technical report, WRL Research Report 89/1.
- C. Gordon Bell and A. C. Newell. 1971. *Computer structures: Readings and examples*. McGraw-Hill.
- Manuel E. Benitez and Jack W. Davidson. 1994. The advantages of machine-dependent global optimization. In *Programming Languages and System Architectures*, pages 105–124.
- Nick Benton, Andrew Kennedy, and George Russell. 1998. Compiling Standard ML to Java Bytecodes. In *International Conference on Functional Programming*, pages 129–140.
- Nick Benton, Andrew Kennedy, and Claudio V. Russo. 2004. Adventures in Interoperability: The SML.NET Experience. In *International Conference on Principles and Practice of Declarative Programming*, pages 215–226.
- Mark Bezem, Jan Willem Klop, and eds. Roel de Vrijer. 2003. *Term Rewriting Systems*. Cambridge University Press, Cambridge, UK.
- Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18 (9):807–820.
- Gunnar Braun, Achim Nohl, Weihua Sheng, Jianjiang Ceng, Manuel Hohenauer, Hanno Scharwächter, Rainer Leupers, and Heinrich Meyr. 2004. A novel approach for flexible and consistent ADL-driven ASIP design. In *Conference on Design Automation*, pages 717–722, New York, NY. ACM.
- Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. 1999 (June). The Jalapeño dynamic optimizing compiler for Java. In *Java Grande Conference*, pages 129–141, San Francisco, CA, United States.
- Maurice Castro. 2001. The EC Erlang compiler. In *International Erlang/OTP User Conference*.
- R. G. Cattell. 1980. Automatic derivation of code generators from machine descriptions. *Transactions on Programming Languages and Systems*, 2(2):173–190.

- Roderic G. Cattell. 1982. *Formalization and Automatic Derivation of Code Generators*. UMI Research Press, Ann Arbor, Michigan.
- Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. *Porting MLton*. <http://mlton.org/PortingMLton>.
- Jianjiang Ceng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. 2005. C compiler retargeting based on instruction semantics models. In *Conference on Design, Automation and Test in Europe*, pages 1150–1155.
- Craig Chambers. 1992. Object-oriented multi-methods in Cecil. In *European Conference on Object-Oriented Programming*, pages 33–56, London, UK. Springer-Verlag.
- David Chase. 1992 (June)a. Implementation of exception handling (part one). *Journal of C Language Translation*.
- David Chase. 1992 (June)b. Implementation of exception handling (part two). *Journal of C Language Translation*.
- David R. Chase. 1987. An improvement to bottom-up tree pattern matching. In *Symposium on Principles of Programming Languages*, pages 168–177. ACM Press.
- C. Cifuentes, B. Lewis, and D. Ung. 2002. Walkabout – a retargetable dynamic binary translation framework. Technical report, Sun Microsystems.
- Christian S. Collberg. 1997. Reverse interpretation + mutation analysis = automatic retargeting. In *Conference on Programming Language Design and Implementation*, pages 57–70.
- Todd A. Cook, Paul D. Franzone, Edwin A. Harcourt, and Thomas K. Miller III. 1993. System-level specification of instruction sets. In *International Conference on Computer Design*, pages 552–557.
- Keith D. Cooper and Linda Torczon. 2004. *Engineering a Compiler*. Morgan Kaufmann Publishers, San Francisco, California.
- Jack W. Davidson. 2008a. Personal communication.
- Jack W. Davidson. 2008b. Personal communication.
- Jack W. Davidson and Christopher W. Fraser. 1980. The design and application of a retargetable peephole optimizer. *Transactions on Programming Languages and Systems*, 2(2):191–202.
- Jack W. Davidson and Christopher W. Fraser. 1984. Code selection through object code optimization. *Transactions on Programming Languages and Systems*, 6(4):505–526.

- Jack W. Davidson and David B. Whalley. 1991. Methods for saving and restoring register values across function calls. *Software Practice and Experience*, 21(2):149–165.
- João Dias and Norman Ramsey. 2006. Converting intermediate code to assembly code using declarative machine descriptions. In *Symposium on Compiler Construction*.
- Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N. J.
- Danny Dubé and Marc Feeley. 2005. Bit: A very compact scheme system for microcontrollers. *Higher Order Symbolic Computing*, 18(3-4):271–298.
- Jon Eddy. 2002. A continuation-passing operator tree for pattern matching. Harvard College senior thesis.
- Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. 1989. BEG – a generator for efficient back ends. In *Conference on Programming Language Design and Implementation*, pages 227–237.
- Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006 (November). XFI: Software guards for system address spaces. In *Symposium on Operating System Design and Implementation*, pages 75–88, Seattle, WA.
- Stefan Farfeleder, Andreas Krall, Edwin Steiner, and Florian Brandner. 2006. Effective compiler generation by architecture description. In *Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 145–152.
- A. Fauth, J. Van Praet, and M. Freericks. 1995. Describing instruction set processors using nML. In *EDTC '95: Proceedings of the 1995 European conference on Design and Test*, page 503, Washington, DC. IEEE Computer Society.
- Lee D. Feigenbaum. 2001. Automated translation: Generating a code generator. Harvard College senior thesis.
- Mary F. Fernández. 1995. Simple and effective link-time optimization of modula-3 programs. In *Conference on Programming Language Design and Implementation*, pages 103–115.
- Mary F. Fernández and Norman Ramsey. 1997. Automatic checking of instruction specifications. In *Proceedings of the International Conference on Software Engineering*.

- Christopher W. Fraser and David R. Hanson. 1995. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, Boston, Massachusetts.
- Christopher W. Fraser and David R. Hanson. 2003. `src/x86linux.md`. Source code of lcc version 4.2.
- Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. 1992. Engineering a simple, efficient code-generator generator. *Letters on Programming Languages and Systems*, 1(3):213–226.
- Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. 1992. BURG: fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76.
- Nicolas Fritz, Philipp Lucas, and Reinhard Wilhelm. 2007 (October). Exploiting SIMD parallelism with the CGiS compiler framework. In Vikram Adve, María Jesús Garzarán, and Paul Petersen, editors, *Workshop on Languages and Compilers for Parallel Computing*.
- Lal George and Andrew W. Appel. 1996 (May). Iterated register coalescing. *Transactions on Programming Languages and Systems*, 18(3):300–324.
- R. Steven Glanville and Susan L. Graham. 1978. A new method for compiler code generation. In *Symposium on Principles of Programming Languages*, pages 231–254.
- Paul Govereau. 2003. Implementation of tiger compiler targeting C--. Part of C-- distribution.
- Torbjoern Granlund and Richard Kenner. 1992. Eliminating branches using a superoptimizer and the GNU C compiler. *Conference on Programming Language Design and Implementation*, 27(7):341–352.
- John V. Guttag and James J. Horning. 1993. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY.
- George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. 1997. ISDL: An instruction set description language for retargetability. In *Design Automation Conference*, pages 299–302.
- Silvina Hanono and Srinivas Devadas. 1998. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *Design Automation Conference*, pages 510–515.
- Roger Hoover and Kenneth Zadeck. 1996. Generating machine specific optimizing compilers. In *Symposium on Principles of Programming Languages*, pages 219–229, New York, NY. ACM.

- G. Huet and D. S. Lankford. 1978. On the uniform halting problem for term rewriting systems. Technical report, INRIA.
- Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. 2003. Effectiveness of cross-platform optimizations for a Java just-in-time compiler. *SIGPLAN Notices*, 38(11):187–204.
- Cliff B Jones. 1986. *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., Hertfordshire, UK.
- Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: a goal-directed superoptimizer. In *Conference on Programming Language Design and Implementation*, pages 304–314, New York, NY.
- Rajeev Joshi, Greg Nelson, and Yunhong Zhou. 2006. Denali: A practical algorithm for generating optimal code. *Transactions on Programming Language Systems*, 28(6):967–989.
- Gerry Kane. 1989. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ 07632.
- Peter B. Kessler. 1986. Discovering machine-specific code improvements. In *Symposium on Compiler Construction*, pages 249–254, New York, NY.
- Jan Willem Klop. 1992. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press.
- D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. 1995. *Chess : retargetable code generation for embedded DSP processors*, pages 85–102. Kluwer Academic Publishers.
- Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Symposium on Principles of Programming Languages*, pages 42–54, New York, NY. ACM Press.
- Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2004. *The Objective Caml system release 3.09: Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique.
- Tim Lindholm and Frank Yellin. 1999. *The Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

- Christian Lindig and Norman Ramsey. 2004. Declarative composition of stack frames. In *Symposium on Compiler Construction*.
- B. H. Liskov and A. Snyder. 1979. Exception handling in CLU. *IEEE Transactions on Software Engineering*, 5(6):546–558.
- Hans Lunell. 1983. *Code Generator Writing Systems*. Software Systems Research Center, Linköping, Sweden.
- Henry Massalin. 1987 (October). Superoptimizer: A look at the smallest program. *Conference on Architectural Support for Programming Languages and Operating System*, 22(10):122–127.
- Stephen McCamant and Greg Morrisett. 2006 (August 2–4.). Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium*, pages 209–224, Vancouver, BC, Canada.
- Greg Morrisett, David Walker, Karl Cray, and Neal Glew. 1999. From System F to typed assembly language. *Transactions on Programming Languages and Systems*, 21(3):527–568.
- George C. Necula. 1997 (Jan). Proof-carrying code. In *Symposium on Principles of Programming Languages*, pages 106–119, Paris, France.
- George C. Necula. 2000. Translation validation for an optimizing compiler. In *Conference on Programming Language Design and Implementation*, pages 83–94.
- Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Notices*, 42(6):89–100.
- J.D. Oakley. 1979 (April). Symbolic execution of formal machine descriptions. Technical Report TR79–117, Computer Science, Carnegie-Mellon University. Ph. D. dissertation.
- Reuben Olinsky, Christian Lindig, and Norman Ramsey. 2006. Staged allocation: A compositional technique for specifying and implementing procedure calling conventions. In *Symposium on the Principles of Programming Languages*.
- Eduardo Pelegrí-Llopert and Susan L. Graham. 1988. Optimal code generation for expression trees: An application burs theory. In *Symposium on Principles of Programming Languages*, pages 294–308, New York, NY. ACM Press.
- Simon Peyton Jones, Norman Ramsey, and Fermin Reig. 1999. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*.

- Simon L. Peyton Jones. 1992. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202.
- Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166.
- Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. 1996. tcc: A template-based compiler for ‘C. In *Workshop on Compiler Support for Systems Software*.
- Mark Probst. 2001. Proper tail recursion in C. Technical Univeristy of Vienna Diploma Thesis.
- Todd A. Proebsting. 1992. Simple and efficient BURS table generation. In *Conference on Programming Language Design and Implementation*, pages 331–340.
- Norman Ramsey. 2000. Pragmatic aspects of reusable program generators. In *International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 149–171, London, UK. Springer-Verlag.
- Norman Ramsey. 2004. expander.nw. Source code of Quick C-- compiler.
- Norman Ramsey and Cristina Cifuentes. 2003. A transformational approach to binary translation of delayed branches. *Transactions on Programming Languages and Systems*, 25(2):210–224.
- Norman Ramsey and Jack W. Davidson. 1998 (June). Machine descriptions to build tools for embedded systems. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *lncs*, pages 172–188. Springer Verlag.
- Norman Ramsey and João Dias. 2005. An applicative control-flow graph based on Huet’s zipper. In *ML Workshop*.
- Norman Ramsey and Mary F. Fernández. 1997. Specifying representations of machine instructions. *Transactions on Programming Languages and Systems*, 19(3):492–524.
- Norman Ramsey and David R. Hanson. 1992. A retargetable debugger. In *Conference on Programming Language Design and Implementation*, pages 22–31.
- Norman Ramsey and Simon L. Peyton Jones. 2000. A single intermediate language that supports multiple implementations of exceptions.

In *Conference on Programming Language Design and Implementation*.

Kevin Redwine and Norman Ramsey. 2004. Widening integer arithmetic. In Evelyn Duesterwald, editor, *Conference on Compiler Construction, LNCS volume 2985*, pages 232–249. Springer.

Michael D. Smith. 1996. Extending SUIF for machine-dependent optimizations. In *SUIF Compiler Workshop*, pages 14–25.

Michael D. Smith, Norman Ramsey, and Glenn Holloway. 2004. A generalized algorithm for graph-coloring register allocation. In *Conference on Programming Language Design and Implementation*, pages 277–288.

Amitabh Srivastava and Alan Eustace. 1994 (June). ATOM: A system for building customized program analysis tools. *Conference on Programming Language Design and Implementation*, 29(6):196–205.

Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. 1999. Age-based garbage collection. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 370–381.

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.

David Tarditi, Peter Lee, and Anurag Acharya. 1992 (June). No assembly required: Compiling Standard ML to C. *Letters on Programming Languages and Systems*, 1(2):161–177.

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

David Wakeling. 1998. A Haskell to Java virtual machine code compiler. In *International Workshop on Implementation of Functional Languages*, pages 39–52. Springer-verlag.

Henry S. Warren. 2002. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

J.D. Wick. 1975 (August). Automatic generation of assemblers. Technical report, Computer Science, Yale University. Ph. D. dissertation.

Jeannette M. Wing. 1983 (May). A two-tiered approach to specifying programs. Technical Report LCS/TR–299, Department of Electrical Engineering and Computer Science, MIT. Ph. D. Thesis.