Work in progress

Design Principles for Machine-Description Languages

Norman Ramsey, Harvard University

Jack W. Davidson, University of Virginia

Mary F. Fernández, AT&T Research

Abstract

We have taken a federated approach to the design of domain-specific languages that support automatic generation of systems software. These languages are intended to help generate parts of software tools that manipulate machine instructions. This paper explains the goals and principles that have governed the design and implementation of these languages. The primary design goal for languages in the confederation is to ensure that the same machine descriptions can be used to help build a variety of different tools. Other goals include producing useful results from partial descriptions, providing mechanisms that build users' trust in descriptions, and making descriptions concise and readable. Our design and development of machine-description languages has been guided by a dozen language-design principles. Using examples both from our work and from others' work, we describe and illustrate each principle. Some of the principles are well known to the programming-language community and have been previously described in the literature, but many of the principles were not obvious from the beginning and have not been as well described in the literature. These principles have emerged from our goals and our application area. They will interest other researchers who design machine-description languages as well as other designers of domain-specific languages.

1 Introduction

Many innovative programming-environment tools work with machine-code representations of programs instead of the more traditional source-code representations. Such tools include testing tools (Hastings and Joyce 1992), simulators (Cmelik and Keppel 1994; Rosenblum et al. 1997), binary translators (Sites et al. 1993), instruction-level profilers (Ball and Larus 1994), proof-carrying code (Necula 1997), typed assembly language (Morrisett et al. 1998), software fault isolators (Wahbe et al. 1993), link-time optimizers (Benitez and Davidson 1988; Srivastava and Wall 1993; Fernández 1995b), general executable editors (Larus and Schnarr 1995; Romer et al. 1997), and many others. Unfortunately for the programmer who wants to use these tools, the amount and complexity of their machine-dependent code make them difficult to retarget. Most are available for only one or two hardware platforms.

Along with other researchers, we have conjectured that machine-level software tools could be retargeted much more easily if the machine-dependent parts were generated from machine descriptions, rather than written by hand. To this end, we are developing a loose confederation of domain-specific languages. We wish to ensure that the *same* machine descriptions can be used to help build a variety of *different* tools, like those enumerated above. The emphasis on reuse distinguishes our work from previous work, and it has had a strong influence on the design of our languages.

What distinguishes our *confederated approach* to machine description is that information about a machine may be distributed over a number of descriptions, which need not be written in a common language or supported by a single tool. They are tied together only by their use of *shared semantic models* of essential aspects of machines. We have developed two such models; one expresses the algebraic structure of an instruction set, and the other identifies all of the mutable state that could change while a machine is executing a program.

For building systems software, we would like to see a confederation of languages covering such properties of machines as the binary representations of instructions, the assembly language, the semantics of the instructions, the calling conventions, and the microarchitecture. Interpreted strictly according to the criteria in this paper, our confederation currently has only two languages: SLED, which describes representations of machine instructions (Ramsey and Fernández 1997), and λ-RTL, which describes semantics of machine instructions (Ramsey and Davidson 1998). Interpreted more loosely, the Calling Convention Language (Bailey and Davidson 1995) and Milner's (1999) pipeline-description language could also be considered members of the confederation.

This is an experience paper; its contributions are to explain the goals and principles that have governed the design of SLED and λ-RTL and to relate these principles to other machine-description languages.

2 Machine descriptions and their uses

There is a large body of prior work on machine description. Because we wish to relate our design principles not only to our own work, but to this prior work, we begin by discussing some significant and representative machine-description languages.

Machine descriptions have been most successful in building retargetable compilers, but the descriptions used in compilers are hard to reuse, because they typically mix information about the target machine with information about the compiler. For example, "machine descriptions" written using tools like CODEGEN (Aigrain et al. 1984), BEG (Emmelmann, Schröer, and Landwehr 1989), and BURG (Fraser, Henry, and Proebsting 1992) are actually descriptions of code generators, and they depend not only on the target machine but also on a particular intermediate language. In extreme cases (e.g., `gcc`'s `md` files), the description formalism itself depends on the compiler. Our confederated approach separates machine properties from compiler concerns.

Maril, the description language used in the Marion system for retargetable code generation (Bradlee, Henry, and Eggers 1991), uses a hybrid approach. Maril describes each instruction's effect using an assignment expression in the C programming language. The Marion system translates each assignment into an intermediate-representation tree, thereby building a mapping between target-machine instructions and the compiler's intermediate representation; this mapping is necessarily compiler-dependent. Maril can also specify, cycle by cycle, the hardware resources that each instruction needs; this information describes only the machine, not Marion's compiler. By combining the IR mappings with information about hardware resources, the Marion system can support code-generation strategies in which instruction selection, register allocation, and instruction scheduling are tightly integrated.

Some existing languages for machine description, like VHDL (Lipsett, Schaefer, and Ussery 1989) and Verilog (Thomas and Moorby 1995) do describe only properties of machines, but unfortunately they are ill-suited for generating systems software. There are a couple of fundamental reasons why these languages are unsuitable for generating systems software.

First, both Verilog and VHDL were designed to support circuit synthesis. Successful synthesis requires generation of practical, compact circuits. To achieve the goal of efficient circuits with existing synthesis techniques, the "distance" between a programmed specification and circuit realizations must be small. Consequently, these description languages are too low a level, focusing more on pragmatics and less on semantics.

Second, these languages were designed to describe the behavior of a circuit to a sufficient level of detail so that the hardware can be simulated. Indeed, simulation engines are important components of VHDL and Verilog systems. Thus, Verilog and VHDL were designed to specify behavior (i.e., how to do something) rather than to specify semantics (i.e., what is to be done). This emphasis on specification of behavior over semantics is reflected by the procedural nature of VHDL and Verilog. Verilog is based on C while VHDL is based on Ada.

VHDL's and Verilog's emphasis on programmed realizations over declarative specifications make it hard to process them to generate systems software. In principle, one could build tools to read these descriptions and discover architectural-level abstractions from implementation-level information—but the effort involved would be considerable. We believe that such effort would be better invested in describing architectural-level abstractions directly, then showing consistency with lower-level descriptions. Because systems software interacts with a machine through its architecture, we want to formalize what architecture manuals say, not what chip designers know.

Arvind and Shen (1999) shows how to use term-rewriting systems to specify the structure and behavior of a microprocessor. Using this formal technique, one can describe different implementations of a single architecture, e.g., with and without pipelining or register renaming. Such descriptions work well for verification; for example, one can show that different implementations can simulate one another's behaviors. As with VHDL and Verilog, however, the level of abstraction is not well matched to the problem of creating systems software.

The informal notation ISP (Bell and Newell 1971) and both of its formal derivatives, ISPL (Barbacci and Siewiorek 1977) and ISPS (Barbacci and Siewiorek 1982), are venerable hardware-description languages based on register transfers. ISPL and ISPS provide formal notations in which one can write a program that decodes a binary instruction format and creates register transfers, which can then be used for simulation or for circuit design. Descriptions based on ISP are hard to reuse outside the context of translating binary to register transfers, in part because the ISP family lacks suitable abstractions. For example, these languages provide no formal way to identify machine instructions and their operands. ISP also requires more hand-work than later systems; for example, the author of a description has to build an instruction decoder by hand, using only low-level programming constructs.

Although ISP was designed to emit components for simulation and hardware design, Cattell (1980) successfully reuses ISP descriptions to build code generators. The reuse requires an intermediate translation step, which transforms the procedural ISP description into a declarative description that maps machine instructions to the compiler's intermediate-representation trees. Unfortunately, this intermediate step is not fully automated, but requires human assistance (Oakley 1979).

An nML description combines a high-level model of a processor's state with an attribute grammar describing the syntax and semantics of its instruction set (Fauth, Praet, and Freericks 1995). The language uses bit strings and byte strings to describe syntax, and register transfers to describe semantics. A description can be used to generate a simulator or a rudimentary code generator.[1] LISAS (Cook and Harcourt 1994) is another specification language that includes distinct semantic and syntactic descriptions. It specifies binary representations by mapping sequences of named fields onto sequences of bits, a technique that works well for RISC machines, but is awkward for CISC.

The *spawn* tool to support executable editing (Larus and Schnarr 1995) uses machine descriptions to provide machine-independent primitives that query instructions. The syntactic part of a *spawn* machine description is derived from a subset of SLED. The semantic part associates register transfers with particular binary representations. From this combined syntactic and semantic information, the *spawn* tool generates classifiers that put instructions into categories such as jump, call, store, invalid, etc. It finds the registers that each instruction reads and writes, and it generates C++ code to replicate such computations as finding the targets of indirect branches. Additions to *spawn* mix register-transfer semantics with explicit directives that acquire and release pipeline resources; this additional information enables *spawn* to use instruction-level parallelism to hide much of the overhead of adding instrumentation (Schnarr and Larus 1996).

The Instruction Set Description Language (ISDL) takes a different approach to tool generation (Hadjiyiannis, Hanono, and Devadas 1997). ISDL descriptions include fragments of C-like code for assembly or disassembly, and the ISDL processor stitches these fragments together to generate assemblers. These fragments may be written by hand, or they may be generated by another tool. For example, ISDL descriptions contain fragments that compute binary representations by shifting and masking fields, then or-ing the results. In other description frameworks, it is possible to derive such fragments from descriptions that are written at higher levels of abstraction. ISDL is planned to support generation of code generators and simulators, with a focus on VLIW machines.

The optimizer PO (Davidson and Fraser 1984) and its derivatives, gcc (Stallman 1992) and vpo (Benitez and Davidson 1994), use machine descriptions to retarget optimizers. All optimizations are performed on machine instructions represented as Register-Transfer Lists, or RTLs (Davidson 1981). The two key machine-dependent components are a translator and a recognizer of RTLs. The translator is used to translate an RTL to an equivalent machine instruction, typically written in assembly language. The recognizer is used by the system to ensure that as RTLs are created and modified, each RTL is representable by a target machine instruction. The recognizer is also used to realize a peephole optimizer, which replaces the conventional code generator in some systems.

In PO, the machine description states a one-to-one, bidirectional correspondence between RTLs and assembly language. Because the descriptions are oriented towards producing recognizers and translators, the emphasis is less towards precise semantics of instructions and more towards writing patterns for identifying instructions. This lack of semantic precision limits reuse. In later systems, the "machine description" is actually a direct encoding (e.g., using Yacc and C) of a recognizer/translator. It is not possible to reuse these descriptions. Still, this approach to producing retargetable optimizing compilers has proven to be effective and adaptable.

Bailey and Davidson (1996a) proposes a machine-description framework that lies somewhere between a monolithic description language and our confederated approach. The Computer Systems Description Language, or CSDL, is to be an *extensible* machine-description language that will support both reusable and application-dependent parts. A distinguishing feature of the proposal is a linking mechanism that can be used to share information among parts or to associate particular parts with particular applications.

Two things distinguish our work from this proposal. CSDL requires that every description include a "core" part, identified as "the description of the instruction set of the machine." The examples suggest that what is meant by this core is a description of the *semantics* of the instruction set, given at a register-transfer level. The instruction-set model shared by members of our confederation is much lighter weight, as it requires only knowledge of the algebraic structure of the

[1] Code generators created from nML descriptions do not support procedure calls.

instruction set (what operands can be used with what instructions), not any knowledge of semantics.

The other distinction is that our approach does not require all description languages to work with a common tool that links information between descriptions. The benefit of our simpler, confederated approach is that we have much more flexibility in implementing tools to support various description languages. The cost is that we have no way of addressing the concern that the same information might be specified redundantly, or even worse, inconsistently, in two different descriptions.

Despite the distinctions, there are many points of similarity between our approach and the CSDL approach, and there is every reason to believe that Bailey and Davidson's (1995) Calling Convention Language would fit nicely into our confederation.

3 Design goals

As SLED, λ-RTL, and our shared models evolved, we developed certain principles for their design. Many of these principles were not obvious from the beginning, but emerged during the design process. The main contribution of this paper is to discuss these principles, so that designers of other domain-specific languages can see whether and when to apply them. The paper answers the classic question, "what do you know now that you wish you had known at the beginning of the project?"

Our design principles depended on our overall goals, and they might not apply to projects with other goals. We begin, therefore, by presenting these goals, with higher-priority goals first.

Reusable descriptions It should be possible to reuse the *same* machine descriptions to generate parts of many *different* software tools, and even different *kinds* of software tools. This was our primary design goal—it is what distinguishes our work from other work on machine description. As corollaries, description languages should apply to a variety of machines, and accompanying tools should support a variety of implementation languages for applications.

Meaningful partial descriptions The description framework should make it possible to write down only the information that is needed for a particular task. For example, someone adapting a dynamic optimizer (Bala, Duesterwald, and Banerjia 2000) into a general a framework for binary rewriting might begin by working only with instructions that affect control flow. While it may be harder to reuse partial descriptions, we felt that our framework had to produce useful components without requiring a "complete theory of machine M," lest the startup cost for using the framework be too high.

Reliable descriptions The description framework should have mechanisms to help give the author confidence that what he wrote was what he intended. Such mechanisms probably include a mix of language features, compile-time analyses, and testing. The value of a mechanism should be judged by considering what kinds of errors the mechanism can prevent or detect.

Usable languages, readable descriptions Domain experts should be able to read and understand existing descriptions and to use the description framework to create new descriptions. People who are not experts in the domain should be able to understand at least parts of existing descriptions, although they may not be able to write new ones.

Concise descriptions Machine descriptions should be concise. One line of description per instruction seems ideal; one page of description per instruction is far too much. Brevity often enhances readability, but in cases where the two are in tension, a *language* should not favor one or the other, but should enable its *users* to strike an appropriate balance.

Making descriptions readable and concise contributes to the confidence an author can place in them, so it would be possible to treat these not as goals but as means of achieving the goal of reliability. We felt, however, that both readability and brevity were such valuable properties that they deserved to be goals in themselves, and we appealed to them directly during the design process.

Design goals have changed somewhat in 25 years. Barbacci and Siewiorek (1977) cites readability, completeness, flexiblity, and brevity as essential goals for ISP. While we agree on the need for readability and brevity, we have relaxed the requirement of completeness, as we have found in practice that incomplete descriptions can be useful for generating systems software. The most significant difference between our goals and the goals of ISP is that we insist on reusable descriptions. The need for reuse has influenced our designs throughout.

4 Sketches of our shared models, SLED, and λ-RTL

We illustrate our design principles with examples both from our work and from others' work. To make the illustrations clear, we sketch not only the models used in our confederation, but also two of the confederated languages: SLED and λ-RTL.

The confederation's shared models

We developed our models based in part on study of the descriptions used to help build a variety of systems software. This software included an optimizer (Benitez and Davidson 1988), a debugger (Ramsey and Hanson 1992), an instruction scheduler (Proebsting and Fraser 1994), a call-sequence generator (Bailey and Davidson 1995), a linker (Fernández 1995a), and an executable editor (Larus and Schnarr 1995). All the descriptions refer either to a machine's instruction set or to the locations that store the machine's state. For example, the descriptions used by the scheduler and linker refer only to the machine's instructions and the properties thereof. The descriptions used in the call-sequence generator and in the debugger's stack walker refer only to storage locations, explaining in detail how values move between registers and memory. Some descriptions, like those used in the optimizer and the executable editor, refer both to instructions and to state, and in particular, they show how the execution of instructions changes the machine's state.

Given these observations, we have built our confederation around the models of machine instructions and machine state that we present below. Every language in the confederation refers to one or the other, and some languages refer to both.

Modelling instructions

We model an *instruction set* as a list of instructions; each instruction is a *constructor* that is applied to operands. Our languages distinguish the *instruction*—the constructor itself—from an *instance* of the instruction—the result of applying the constructor to operands. Both names and types of operands are significant. Operand types include integers of various widths; it is also possible to introduce new types to define such machine-dependent concepts as effective addresses. Values of these new types are created by applying suitable constructors. Introducing such types helps keep models from getting too big.

For example, we introduced the type `Address` to describe the addressing modes of the SPARC. The SPARC's two addressing modes are represented as the constructors `indexA` and `dispA`, both of type `Address`. The operands of `indexA` are the 5-bit integers `rs1` and `rs2`; the operands of `dispA` are `rs1` and `simm13`, which is a 13-bit integer. Starting from the bottom, `rs1`, `rs2`, and `simm13` represent integer operands; they can be passed to constructors `indexA` and `dispA` to produce operands of type `Address`, which finally can be passed to constructors such as `ld` or `st` to build instructions.

In our model, an instruction set is isomorphic to an algebraic datatype in a functional language such as ML (Milner et al. 1997) or Haskell (Peyton Jones et al. 1999). Our model is also very like a grammar, in which the start symbol is "instruction" and new constructor types correspond to interesting nonterminals. Researchers working directly with grammars have also found it necessary to introduce interesting nonterminals; for example, a Graham-Glanville grammar without such nonterminals would require over 8 million productions to describe the VAX (Graham, Henry, and Schulman 1982).

Modelling state

Our confederated languages represent state as the contents of *storage spaces*. A storage space is a sequence of mutable cells. Each cell contains a bit vector. As in an array, cells are all the same width, and they are indexed by integers. Storage spaces model not only main memory and general-purpose registers, but also special-purpose registers, condition codes, and so on. The entire state of a machine can be described as the contents of its storage spaces. Our languages refer to cells by naming the storage space and giving an integer expression identifying a cell within that storage space. For example, on the Pentium, `$r[0]` refers to general-purpose register 0, i.e., register EAX.

SLED: Syntax of instructions

SLED (the Specification Language for Encoding and Decoding) is a member of our confederation that describes the assembly-language and binary representations of machine instructions (Ramsey and Fernández 1997). It refers only to instructions, not to state.

SLED models a binary representation as a sequence of *tokens*, which are bit vectors. Tokens may represent whole instructions, as on RISC machines, or parts of instructions, as on CISC machines. For example, the Pentium's opcode byte, its "Mod R/M" byte, and its three sizes of displacements are modelled as tokens of different *classes*.

In SLED, each class of token is declared with multiple *fields*, which are contiguous ranges of bits within tokens of that class. SLED's fields are the fields that are used informally in architecture manuals. A `fields` declaration binds field names to bit ranges and specifies the number of bits in tokens of its class.

```
fields of itoken (32)
  op 30:31 rd 25:29 op3 19:24 rs1 14:18
  i 13:13 simm13 0:12 opf 5:13 rs2 0:4
  op2 22:24 imm22 0:21 a 29:29 cond 25:28
  disp22 0:21 asi 5:12 disp30 0:29
```

As the example suggests, tokens of the same class can be partitioned into fields in more than one way. For example, the SPARC call instruction partitions its 32-bit token into only 2 fields (`op` and `disp30`), whereas the floating-point add instructions use 6 fields.

SLED uses *patterns* to describe binary representations. Patterns are formed from constraints on fields, such as `op = 0`. Patterns can be conjoined (when constraints apply to fields in the same token), concatenated (when constraints apply to tokens in sequence), or disjoined (when alternative constraints might apply). Typically only conjunction and concatenation are used to describe individual instructions; disjunction is used to describe groups of instructions that share some property of interest, e.g., the property of being "nonbranching instructions." Patterns can be named; they are particularly well suited to describing opcodes, as in this declaration of `add`.

```
patterns add is op = 2 & op3 = 0
```

SLED describes the binary representation of an instruction (constructor) by giving the constructor's name, its operands, and a pattern in which the operands may appear as free variables. In general, a set of equations defines the relationship between an instruction's operands and the fields of its binary representation. For example, in this fragment of SLED, the equation in braces shows how the target address of a call instruction is computed by sign-extending (`!`) the `disp30` field, multiplying by 4, and adding the result to the address of the instruction.

```
constructors
  call addr { addr = L + 4 * disp30! } is
    L: CALL & disp30
```

On the last line, `disp30` is syntactic sugar for the constraint `disp30 = disp30`, which means "the value of the field `disp30` (left-hand side) is equal to the value of the operand `disp30` (right-hand side)."

Such syntactic sugar is useful because in almost all cases, the fields of an instruction are directly equal to the values of the operands with the same name, and no equations are needed. For example, these declarations define the binary representations of the SPARC's "register or immediate" operands.

```
constructors
 rmode rs2     : reg_or_imm is i = 0 & rs2
 imode simm13! : reg_or_imm is i = 1 & simm13
```

On RISC machines, the binary representation of an instruction is usually the conjunction of the opcode's pattern with the bindings of the operands. In this case, the pattern can be omitted from the definition of the constructor.

```
constructors
  add rs1, reg_or_imm, rd
```

Finally, patterns bound to disjunctions are expanded on the left-hand sides of constructor definitions, so that binary representations for many similar instructions can be defined at once. Here, the `patterns` declaration associates `alu` with a disjunction of 42 opcodes, and the `constructors` declaration defines assembly-language and binary representations for 42 instructions.

```
patterns
  alu is add | addcc | addx | ...
constructors
  alu rs1, reg_or_imm, rd
```

SLED's constructor definitions include specifications of assembly-language representations. These representations are not specified using any sort of explicit, string-valued expressions; instead, they are inferred from "punctuation" placed with the operands of the instructions. The preceding examples use commas to separate operands; this example shows the square brackets that SPARC assembly language requires addresses in load instructions:

```
patterns loadg is ldsb | ldsh | ldub | ...
constructors  loadg [Address], rd
```

SLED descriptions are very concise, ranging in size from 125-250 lines for machines like the MIPS, Alpha, and SPARC, to about 500 lines for the Pentium.

λ-RTL: Semantics of instructions

λ-RTL is a member of the confederation that describes the semantics of machine instructions (Ramsey and Davidson 1998). It refers to both instructions and state. It describes "semantics" in the simplest possible sense: how executing an instruction changes the state of the machine. λ-RTL assigns to each instance of an instruction a function from storage spaces to storage spaces. Because most instructions change at most a few storage cells, λ-RTL specifies these functions using register-transfer lists (RTLs).

A register-transfer list is a list of guarded effects, which are separated by bars (`|`). Each effect represents the transfer of a value into a storage location, i.e., a store operation. The transfer takes place only if the guard (an expression) evaluates to **true**. Most effects have no explicit guards; such effects are deemed to be guarded by **true**. Effects in a list take place simultaneously, as in Dijkstra's multiple-assignment statement; an RTL represents a single change of state. For example, an RTL can specify a swap instruction without introducing temporaries and intermediate states that have no counterpart in the execution of the actual machine.

Locations may be single cells or aggregates of consecutive cells within a storage space. Values are computed by expressions without side effects. Expressions may be integer constants, fetches from locations, or applications of *RTL operators* to lists of expressions.

To describe the semantics of instructions, we attach RTLs to constructors, using λ-RTL *attributes*. A default, unnamed attribute is taken as the "main effect" of an instruction. For example, the main effect of the SPARC `call` instruction is to change the "next PC" to the target address and simultaneously store the current PC in register o7.

```
default attribute of
  call (addr) is nPC := addr | Reg.out 7 := PC
```

The instruction puts addr in nPC, not PC, because the call takes place with a delay of one cycle.

The default attribute of an untyped constructor (instruction) is typically an effect, but other kinds of constructors may have other default attributes. For example, SPARC constructors of type reg_or_imm have default attributes that denote 32-bit values.

```
default attribute of
  rmode (rs2)    : reg_or_imm is $r[rs2]
  imode (simm13) : reg_or_imm is sx simm13 : #32 bits
```

In this example, sx is the sign-extension operator, and the type annotation on the right is needed to specify that simm13 is to be extended to 32 bits, not to some other size.

As in SLED, the preceding definitions make reg_or_imm available for use as an operand to another instruction.

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

In a SLED specification, fields declarations determine the sizes of simple operands. In λ-RTL, those sizes must be declared explicitly.

```
operand [rd rs1 rs2] : #5 bits
operand simm13 : #13 bits
operand imm22  : #22 bits
operand addr   : #32 bits
```

λ-RTL's named attributes can be used to attach multiple RTLs to a single constructor. For example, the default attribute of a load instruction might be the transfer of a value from memory into a register. That instruction might also have a trap attribute that tells under what circumstances the instruction traps (e.g., on unaligned loads). Using multiple attributes to define the instruction's behavior makes it easier to write partial descriptions.

```
default attribute of
  ld(address, rd) is $r[rd] := $m[address]
attribute trap of
  ld(address, rd) is
    address mod 4 != 0 --> trap_code := Alignment
```

The trap attribute is a *guarded* effect; if the guard is true, the action of trapping is modelled by assignment to trap_code, just as the action of taking a delayed branch is modelled by assignment to nPC.

These examples show λ-RTL, a notation that is easy for people to read and write, but in which too much information needed by tools is left implicit. Our translator accepts λ-RTL as input and emits "true RTLs," a representation that is fully explicit and designed to be used by tools. To simplify analysis, the representation is a simple, detailed, and unambiguous tree. As much information as possible is explicit; for example, byte order is explicit in every memory reference, and every RTL operator is parameterized by the sizes of its operands and results. We don't care if individual RTLs grow large, as long as they are composed from simple parts using only a few rules. This design choice distinguishes our RTLs from earlier work, which has used smaller RTLs that make implicit assumptions about details like operand widths and byte order. In our confederation, a well-formed RTL has a meaning that is completely independent of the machine(s) being described.

As an example of a true RTL, consider the meaning of a SPARC load instruction using the displacement addressing mode, written in the SPARC assembly language as

```
ld [%sp-12], %i0
```

The effect of this load instruction might be written in λ-RTL as follows:

```
$r[24] := $m[$r[14]+sx(~12)]
```

because the stack pointer is register 14 and register i0 is register 24. The λ-RTL denotes a true RTL that is much more verbose, with the widths of all quantities identified explicitly, as a fully disambiguated tree:

```
                  STORE #32
                 /         \
   AGG B #32 #32            FETCH #32
         |                      |
     CELL 'r'             AGG B #8 #32
         |                      |
     CONST #5                CELL 'm'
         |                      |
        24                 APP ADD #32
                          /           \
                FETCH #32              APP SX #13 #32
                    |                        |
             AGG B #32 #32               CONST #13
                    |                        |
                CELL 'r'                    -12
                    |
                CONST #5
                    |
                   14
```

The constants labeled with hash marks, like #32, indicate the number of bits in arguments, results, or data being transferred. For generating tools, these true RTL trees have one overwhelming advantage; everything is explicit, so we can build tool generators that don't make assumptions about the target machine.

Having seen examples of its input and output, we can characterize the λ-RTL language. The purpose of this language is to support the kinds of register-transfer notations users are already comfortable with (Bell and Newell 1971), but to give the notations a precise meaning in terms of fully explicit RTLs.

λ-RTL is based on typed λ-calculus, but it has a special type system and primitive operators that support the description of RTLs (hence its name). It has many of the features one would expect in a functional language, including first-class functions and anonymous tuples and records, but it does not have algebraic datatypes or recursion. One consequence is that every well-typed expression has a normal form (i.e., all programs terminate).

λ-RTL values have types, and the type system includes special types that help describe machine-level operations. These types include *values*, which are uninterpreted bit vectors, *locations*, which are parts of state that can hold values, and *cells*, which represent the primitive cells in storage spaces. Cells may be aggregated together (with explicit byte order) to form locations. Type checking extends to the sizes of cells, locations, and values; for example, a 16-bit value may not be put in a 32-bit register without an explicit sign-extension, zero-extension, or bit-insertion operator. For example, the semantics of the `imode` constructor given above shows the use of the `sx` sign-extension operator to extend a 13-bit value to 32 bits.

λ-RTL uses a generalization of Milner's (1978) type inference so that types need not be written explicitly, and it uses subtyping so that fetches and aggregations need not be written explicitly. A cell is a subtype of location, so if a cell is used where a location is expected, the translator inserts an aggregation. Similarly, a location is a subtype of value, so if a location is used where a value is expected, the translator inserts a fetch. Nordlander's (1999) subtyping algorithm supports such implicit coercions, without adding subtype constraints to user-defined functions. By eliminating explicit subtyping constraints, this algorithm insulates users from the details of subtyping and gives them a simpler view of the system.[2]

λ-RTL descriptions are relatively concise, although not as concise as SLED descriptions. Because λ-RTL is not as mature as SLED, we can report only on preliminary experience. We have written a full description of the MIPS R4000; 195 lines of λ-RTL describe 125 instructions. λ-RTL can describe 160 SPARC instructions in about 300 lines. The instructions include register windows, control flow, load and store, and all integer and floating-point ALU instructions, including effects on condition codes. The only instructions omitted are some coprocessor instructions, a few privileged load and store instructions, and cache flush. Our partial description of the Pentium is about 130 lines. This description handles the Pentium's addressing modes, the meaning of which is context-dependent, and it describes 42 logical instructions using four different idioms.

[2] A deficiency of Nordlander's algorithm is that it may reject a term because it was unable to figure out where to insert coercions, when another, more general algorithm might have discovered the proper coercions. Worse, it can be difficult to predict which source-language terms can actually be typed by Nordlander's algorithm. But the more general algorithms compute principal types, which can grow very complex, and error messages involving such types are difficult to decode by all but the most experienced experts. We have therefore opted for simplicity over expressive power. Perhaps because the functions used to describe machines are simple, in practice we have never encountered a term in which we had to insert an explicit coercion.

5 Design principles

Section 3 presents our design goals and compares them with other authors' goals. Here we present the design principles we developed with SLED and λ-RTL. Although we phrase our goals as properties we want descriptions to have, and we phrase our principles as imperatives to the language designer, the distinction between goals and principles is not so clear cut. We hope that applying the principles may help designers achieve the goals, but perhaps at some level the distinction is arbitrary.

As with the goals, we present the most important principles first.

Use an appropriate language for each sub-domain

Most machine-description frameworks attempt to describe all properties of interest in one language. The most unusual aspect of our approach is that we use a confederation of languages, not a single language. According to the principle, each language should be tuned to describe a different property of interest. Of course, it is really the semantic model underlying the language that should be tuned, so the language's abstractions will match both the problem domain and the algorithms used by the language's tools. For example, CCL describes state machines, which its tools use to generate calling sequences and test sequences (Bailey and Davidson 1995; Bailey and Davidson 1996b). SLED uses fields and patterns, which its tools use to generate efficient encoders and decoders. λ-RTL uses RTL trees, which its tools use to create or match application-dependent trees and strings. PLUNGE uses datapath graphs, which its tools use to map RTL trees to resource vectors (Milner 1999).

Using multiple languages contributes to each of our design goals. It eases reuse of machine descriptions because a description covering only one sub-domain can be understood and reused without reference to other sub-domains. *Every* description is a partial description; only by collecting a group of descriptions, possibly written in different languages, is it possible to gather all of the available information about a machine. Authors can have more confidence in descriptions because in studying them, they can focus on the one property being described. Descriptions can be made more concise, for two reasons. When only one property is being described, it is easier to "factor out" similarities among multiple instructions and so to write smaller descriptions. Also, special-purpose languages can eliminate notation; for example, where nML requires explicit attribute equations for assembly-language syntax and binary representations, SLED identifies binary and assembly representations without notation like "`image =`" or "`syntax =`." Finally, the individual description languages can be more readable and easier to use, because each one can be tuned to its proper sub-domain. These benefits shouldn't be surprising, because writing multiple descriptions of one machine isn't a radical step; it is separation of concerns, or modularity.

The nML, LISAS, ISDL, ISP, and TOAST (Hoover and Zadeck 1996) frameworks all take the opposite, monolithic approach. Monolithic descriptions can be large; these languages typically require 10–20 lines of description per instruction as opposed to the 1–2 lines required by SLED and λ-RTL. The monolithic approach can make descriptions harder to understand; for example, the ISDL manual warns users not to divide descriptions into separate files, because the effects of different sections cannot be understood in isolation (Hadjiyiannis 1998).

Sometimes a single language can cover multiple sub-domains. For example, SLED describes *two* properties of each instruction: assembly-language syntax and binary representation. As discussed below, using defaults effectively mitigates the damage; it is always possible to write a SLED specification that focuses exclusively on one of the two properties, because the semantics of SLED guarantee a sensible default for the other. *Spawn* also combines two sub-domains; it interleaves descriptions of semantics and pipeline behavior. Defaults make it possible to omit pipeline information without destroying the value of the semantics.

It is not clear how this principle affects the *implementation* of domain-specific languages. Using different languages makes it easier to keep languages small and simple, which in turn makes it easier for third parties to reuse descriptions by creating new implementations of existing languages. For example, Michael and Appel (2000) describes a new implementation of SLED that maps binary representations into "decoding relations" in a formal logic. But using different languages makes it harder to generate machine-dependent code for which information is needed from more than one sub-domain. For example, we currently generate code that maps directly from binary representations to RTLs, for use in a binary translator. Because the SLED and λ-RTL tools are fairly closely integrated, generating this code is straightforward, but there are obvious disadvantages to building a large, monolithic tool that understands all the languages in the confederation. How to build separate tools that can cooperate when necessary is a topic for future research.

Avoid a preferred direction

Both λ-RTL and SLED map between abstract constructors and more concrete properties like binary representation, register-transfer semantics, etc. The most important design principle for both λ-RTL and SLED is that the descriptions should not be biased towards either direction of this mapping. It was therefore an absolute requirement that the *same* parts of the description serve both to map from constructors to a concrete representation (e.g., binary code) and in the opposite direction. To prefer one direction would be to risk being unable to support software tools that need to go in the other direction, or in both directions. To use separate constructs for the two directions would be to risk inconsistency.

This principle, while the most important, may also have been the most difficult to accommodate in the design of SLED. It made us introduce equations mapping between the fields of the binary representation and the operands of the abstract representation. Developing a suitable equation solver required nontrivial extensions to existing techniques (Ramsey 1996).

This principle is also not trivial to apply to λ-RTL. One direction—from instructions to RTLs—is easy. The opposite direction requires that the translator generate a recognizer. This problem is closely related to the problem of match compilation in functional languages, which has been studied extensively (Cardelli 1984; Augustsson 1985; Baudinet and MacQueen 1985; Laville 1991; Maranget 1992).

The retargetable peephole optimizer PO (Davidson and Fraser 1980) exploits this principle; it uses a single description to drive translations between assembly language and ISP-like register transfers, in both directions. Some other machine-description languages seem to have ignored this principle. For example, both ISP and *spawn*/EEL support only instruction decoding; encoding requires a separate assembler.

Sometimes a preferred direction can be overcome by thorough analysis. ISP is designed to map from binary representation to semantics, but Wick (1975) describes analyses that classify variables in an ISP description, recover a high-level, symbolic abstraction, and emit a translator from that abstraction to the binary—an assembler. Because ISP is biased towards the opposite direction, the analyses are complex; for example, they include symbolic execution of register transfers.

This principle militates against designs that include "semantic actions" written in a general-purpose programming language, as in Yacc or BURG (Fraser, Henry, and Proebsting 1992). Such semantic actions necessarily work in one direction only; arbitrary C code, for example, cannot be "inverted."

Analyze at compile time

Compile-time analysis is a significant way to give authors confidence in the correctness of their machine descriptions. While designing our languages, we have considered what kinds of errors might be detected by compile-time analysis. The best kinds of errors are those that can be detected at compile time and that are definitely and unambiguously wrong. A machine description containing such an error can be rejected at compile time as inconsistent. This language-design principle says to get as many errors as possible into this category.

For example, if one uses bit strings to describe representations of instructions, it is possible to get bits out of order, to omit bits, or to duplicate bits. It's hard to imagine being able to check anything other than the length of the bit string, and this check can detect only a single omission or duplication error, and cannot detect even one out-of-order error. By using a model based on fields and tokens, a SLED processor can detect all of these errors. As another example, because SLED distinguishes different token classes even when they have the same size, a SLED compiler can reject, as inconsistent, a binary representation that mixes the `mod` field of the Pentium's Mod R/M byte with the `index` field of its SIB byte.

In the design of λ-RTL, we were concerned about potential errors in specifying the way values are "widened." For example, in a load-halfword instruction, a 16-bit value might be put in a 32-bit register. At least three alternatives are plausible: sign-extend the value, extend it by putting zeroes into the high bits, or insert the value into the least significant 16 bits of the register, leaving the most significant 16 bits unchanged. So when a narrow value is assigned to a wide location, λ-RTL does not choose a default, which might be silently and undetectably wrong. Instead, λ-RTL treats the assignment as a compile-time error.

The next best kinds of errors are those that can be detected at compile time, and are usually wrong, but may in rare cases be correct. We call a description containing such errors *implausible*, because while in principle it could describe an actual machine, in practice one is unlikely to design a machine as described. Our translators issue warnings for implausible descriptions. For example, in SLED, it is implausible that the binary representation of an instruction wouldn't depend on the values of the instruction's operands. Another example of an implausible specification is one that assigns a single binary representation to more than one instruction ("opcode non-uniqueness"). In the current implementation of SLED, we neglected to check for this implausibility, but adding this check would be relatively easy.

In SLED and λ-RTL, the names of operands are significant. This feature, which is not found in general-purpose languages, makes it easier to detect some implausibilities. In λ-RTL, for example, it is implausible that operand `rd` would designate a destination register in almost all instructions but a source register in a few instructions. In SLED, it is implausible that the operand `rd` would be the rightmost operand of almost all instructions but the leftmost operand of a few instructions.

The worst kinds of errors are those that can't be detected at compile time. We call a description *plausible* if it could reasonably be the description of some machine. A plausible description might still contain an error, and whether it actually describes the machine that is intended can be detected only by testing against an external standard, like a vendor's assembler or the chip itself (Fernández and Ramsey 1997). For example, a description that puts destination operands on the left, when they should be on the right, is plausible. So is a description that computes the addresses of branch targets relative to the addresses of the branch instructions, when they should be computed relative to the addresses of the successor instructions.

In addition to enabling early detection of errors, careful language design can also make it easier to perform the analyses that are required for building systems software. For example, vpo, LISAS, and several other tools permit "overlapping register sets," in which two apparently different registers can be aliased to the same location. This device make it possible to work with machines in which the same registers can be used alone or as part of register pairs, for example. The cost, however, is more complicated alias analysis. λ-RTL uses a different mechanism, explicit aggregation, which supports register pairs as well as memory loads and stores of different sizes (e.g., byte, halfword, and full word). As a result, aliasing in λ-RTL is purely notational; the λ-RTL translator can eliminate apparent aliasing, making the resulting register transfers easier to analyze. λ-RTL semantics supports the full generality of aliased registers while simplifying the analysis of machine descriptions.

Help manage repetition

Perhaps the most salient characteristic of machine instructions is the degree to which properties can be almost the same from instruction to instruction. Both SLED and λ-RTL use a repetitive construct in which a list of names on the left is bound to a list of values on the right, where the values on the right may be listed explicitly, but are more often created by some sort of "generating expression." Generating expressions are modelled on expressions in the Icon programming language, which can produce more than one value (Griswold and Griswold 1996). For example, in SLED, the following declarations bind names of opcodes to (partial) binary representations from the opcode tables in the SPARC manual (SPARC 1992):

```
patterns
 [TABLE_F2 call TABLE_F3 TABLE_F4]
    is op = {0 to 3}
patterns
 [ add    addcc    taddcc     wrxxx
   and    andcc    tsubcc     wrpsr
   or     orcc     taddcctv   wrwim
   xor    xorcc    tsubcctv   wrtbr
   sub    subcc    mulscc     fpop1
   andn   andncc   sll        fpop2
   orn    orncc    srl        cpop1
   xnor   xnorcc   sra        cpop2
   addx   addxcc   rdxxx      jmpl
   _      _        rdpsr      rett
   umul   umulcc   rdwim      ticc
   smul   smulcc   rdtbr      flush
   subx   subxcc   _          save
   _      _        _          restore
   udiv   udivcc   _          _
   sdiv   sdivcc   _          _
 ]  is
   TABLE_F3 & op3 = { 0 to 63 columns 4 }
```

As in Icon, operations like field binding and conjunction distribute over the values returned by the generating expression. On the left, the underscores are wildcards, which indicate that the corresponding values of the generating expression should be thrown away, not bound to names. The repetition construct helps keep descriptions concise; here we have specified 55 opcodes in 21 lines. The construct also makes it easy to create descriptions that resemble the "opcode tables" found in architecture manuals, which makes descriptions readable and helps an author gain confidence in them.

λ-RTL uses a similar construct. In λ-RTL, generating expressions may include lists of items in brackets. For example, the following line defines the three functions `andn`, `orn`, and `xnor`, which specify the main operation performed by the SPARC's "bitwise operation with complement" instructions.

```
fun [andn orn xnor] (a, b) is [and or xor](a, com b)
```

When describing the semantics of the instructions themselves, we must account for possible changes to condition codes. The `logical` function specifies the behavior of all the logical instructions, using `set_cc` (not shown here) to set the condition codes when necessary.

```
fun logical (operator, rs1, r_o_i, rd, {set_codes})
  is let val result is operator($r[rs1], r_o_i)
     in  $r[rd] := result |
         set_codes --> set_cc(result, 0, 0)
     end
```

Finally, we use the λ-RTL grouping construct to bind the semantics to the appropriate constructors.

```
default attribute of
 [and or xor andn orn xnor]^[cc ""]
                          (rs1, reg_or_imm, rd)
is
 logical([and or xor andn orn xnor],
         rs1, reg_or_imm, rd,
         {set_codes is [true false]})
```

Using two sets of brackets corresponds to one loop nested within another. This sort of factoring can produce very concise descriptions—here, twelve instructions in one—but the particular notation we have chosen may be difficult to understand, and we provide syntactic alternatives for expressing this kind of repetition.

The factoring technique benefits from our use of different languages for different sub-domains. The natural factorings of semantic information are different, and more fine-grained, than the natural factorings of binary representations. By treating each in its own language, we can exploit all available opportunities for factoring, instead of only those opportunities that are the same for both semantics and representation.

The *spawn* language adapts and improves upon SLED's generating expressions for specifying syntax. For specifying semantics, however, it uses the same square-bracket notation with different semantics. While SLED, λ-RTL, and Icon implicitly distribute operations over values listed in brackets, *spawn* treats these bracketed items as vectors, and *spawn* includes built-in operators that either distribute application of a function over the elements of a vector, or that apply a vector of functions to a vector of values, pairwise. This interpretation of the brackets allows finer control over the way brackets are used, at the cost of using the same notation with different meaning in different parts of the description.

One reason that ISP descriptions are longer than SLED and λ-RTL descriptions is that ISP provides insufficient support for managing repetition. ISP provides first-order functions, which can be used to avoid repeating common computations, like address calculations, but ISP doesn't have anything analogous to generating expressions. Descriptions of the PDP-11, Interdata 8/32, and IBM 370 are 1445, 2345, and 2132 lines of ISPL. Barbacci and Siewiorek (1977) ascribes these sizes to the sheer number of instructions, not to the complexity of the architectures.

Generating expressions capture some, but not all, of the repetitive character of semantic descriptions. Higher-order functions are recognized as ways of defining "patterns of behavior" that can be used over and over; especially in pure languages, they can perform computations that would require new control structures in imperative languages (Hughes 1989; Peyton Jones, Meijer, and Leijen 1998). Higher-order functions can be used in λ-RTL to help avoid repeating similar behaviors in a machine's semantics. For

On the Pentium, the denotation of an effective address depends on the context in which it is used. For example, register 5 denotes `AH`, `SP`, or `ESP`, when it is used in a byte, word, or doubleword context, respectively. A natural representation for such an effective address seems to be a record of type `{b : #8 loc, w : #16 loc, d : #32 loc}`. When an effective address refers to a location in memory, the `M` function uses explicit, little-endian aggregation (`RTL.AGGL`) to produce a record of the right type:

```
fun M address  is
  { b is RTL.AGGL #8 #8  $m[address]
  , w is RTL.AGGL #8 #16 $m[address]
  , d is RTL.AGGL #8 #32 $m[address]
  }
```

A context can be represented by a function that is applied to the record:

```
fun b {b, w, d} is b
fun w {b, w, d} is w
fun d {b, w, d} is d
```

One might expect that such functions could be passed to other functions, like the one that specifies the "two-address" effect:

```
fun llr (left, op, right) size is
  size left := op(size left, size right)
```

This definition of `llr` works, but it has undesired consequences, owing to a well-known limitation of the Hindley-Milner type system. The parameter `size` must have a monomorphic type, like $\{\mathtt{b} : \alpha, \mathtt{w} : \beta, \mathtt{d} : \gamma\} \to \alpha$, and once α is unified with a location (because `size left` is assigned to), it cannot also be unified with a value. This means if `left` is a location, `right` cannot be a value!

We work around this problem by using a *pair* of functions to represent a context. One element of the pair is applied to locations and the other to values:

```
fun llr (left, op, right) size is
  let val (sl, sr) is size
  in  sl left := op(sl left, sr right)
  end
```

It is difficult to explain the need for this transformation to programmers who are not intimately familiar with Milner's type inference and its limitations.

Figure 1: Using λ-RTL to describe the Pentium's context-dependent effective addresses and two-address instructions.

example, Figure 1 shows the development of the `llr` function, which encapsulates both the two-address nature of most Pentium instructions and the context-dependent meanings of their operands. Given `llr`, describing the semantics of some instructions is easy.

```
default attribute of
  ANDmrb (addr, reg) is llr (addr, and, R reg) b
```

Here `addr` is an effective address, and `R` is a function that converts the register number `reg` into the λ-RTL triple that represents an effective address. Because of the technical problem explained in Figure 1, `b` is actually a pair of functions, each of which extracts byte-context meanings from triples.

Higher-order functions are even more powerful when combined with generating expressions. This declaration defines the main effects of 9 constructors.

```
default attribute of
  [AND OR XOR]^mr^[b ow od] (addr, reg) is
    llr (addr, [and or xor], R reg) [b w d]
```

This construct certainly helps one write concise machine descriptions, and it may improve confidence, since if any one of the 9 descriptions is right, they are probably all right. There may be other advantages; Andrew Appel has suggested that factoring the descriptions may help in factoring proofs of properties of the instructions described.[3] It remains to be seen whether such aggressively factored descriptions will be useful or confusing in practice; some readers may find it easier to understand longer descriptions that use fewer generating expressions. By providing suitable mechanisms in the language, we have left that choice to the users.

Make abstractions match the problem

The better a language's abstractions match its problem domain, the easier it is to make the language readable, and the easier it is to analyze descriptions at compile time. For example, nML describes binary representations using bit strings. The basic operations are `format`, which is analogous to `printf`, and concatenation. For example, the encoding of a load instruction might be specified as follows:

```
opn load(r:regm,a:adrmode)
  image=format("00%b%b",r.image,a.image)
```

The bit-string abstraction matches the problem domain, but weakly. It offers little protection against errors: any bit strings can be concatenated, in any order, and all the compiler can do is check the length of the result.

Using SLED's patterns, the description above might be written

```
constructors
 load regm, adrmode is op = 0 & regm & adrmode
```

[3] Private communication, November 1999.

SLED's abstraction does a better job preventing errors. For example, it is impossible to "concatenate the fields in the wrong order," because the positions of the fields are defined when the fields are declared. Since common fields like register operands are defined once but used dozens of times, SLED provides many fewer opportunities for making errors. It is similarly impossible to "include the same bits twice" in the binary representation. A SLED expression like `op = 0 & op = 0` is equivalent to `op = 0`; an expression like `op = 0 & op = 1` is inconsistent and is rejected with an error message.

Finally, SLED's abstraction can describe binary representations with "holes," which are common on modern architectures. For example, the SPARC opcode for single-precision floating-point negate is given by constraining three fields.

`op` = 2	—	`op3` = 52	—	`opf` = 5	—
31 30	29 25	24 19	18 14	13 5	4 0

This opcode can be expressed in SLED as `op = 2 & op3 = 52 & opf = 5`. It cannot be expressed in nML. The same mechanisms enable SLED to express instructions in which the values of parameters are split and placed in noncontiguous parts of the binary representation.

The model of instructions used in SLED and λ-RTL is completely abstract and symbolic, which helps users separate semantic concerns from bit-fiddling. Both nML and LISAS allow some explicit decoding (hand-written switch statements that scrutinize the binary representation) to leak into semantic specifications. Because the abstractions in nML and LISAS do not separate representation from semantics as cleanly as one might like, it is harder to build efficient encoders and decoders automatically.

There are not many different kinds of abstractions that can be used to describe binary representations. To describe semantics, however, we have more choices. Barbacci (1975) divides these choices into three broad levels. A high-level *behavioral* description takes a black-box approach, describing only input/ouput behavior. A *functional* description exposes components of a machine, but describes their relationships abstractly, using functions or algorithms. At the lowest level, a *structural* description describes the real hardware components and their interconnections. Different levels of descriptions are useful for different purposes. The behavioral level is a good fit for describing the programmer's view—the machine architecture. The structural level is a good fit for describing the implementor's view—the machine organization. Because our purpose is to generate systems software, we need abstractions at the behavioral level.

One reason it is hard to reuse machine descriptions from compilers is that in many compilers, the abstraction is not "the semantics of the instructions" but "the relationship of the instructions to my intermediate form." Code-generator generators like BEG, BURG, and Marion work from descriptions of such relationships. This choice of abstractions not only limits reuse, but may make it impossible to describe a full instruction set. For example, because the Marion system can describe only instructions representable by a single assignment, it cannot describe arithmetic instructions that change condition codes, auto-increment addressing modes, swap instructions, etc. (Bradlee, Henry, and Eggers 1991).

The applications of the *spawn* description language illustrate the value of the "matching abstractions" principle. *Spawn* has no abstract model of instructions; instead, it treats a machine instruction as a collection of fields, all of which are available for use in any specification. This treatment is suitable for RISC machines, on which instructions occupy one word and fields are always at known locations. But it is unsuitable for CISC machines, on which instructions vary in size and the location of one field may depend on the value of another. *Spawn* has been used to describe several RISC machines, but it seems difficult, if not impossible, to write a *spawn* description for the x86 or 68000 families.

λ-RTL is another language in which not all abstractions match the problem. λ-RTL is not missing any crucial abstractions, but it contains extra, higher-level abstractions—records, tuples, and functions—that have no obvious counterparts in the problem domain. These abstractions help authors manage repetition in semantic descriptions, as shown above, but functions in particular are *not* well matched to the problem domain, and as a result, it is too easy to write λ-RTL descriptions that are hard to understand. For example, an earlier version of our SPARC description did not pass a Boolean `set_codes` flag; instead, it passed a *code-setting function* that either set the condition codes or did nothing. This version was quite a bit harder to understand, as the effects were distributed over many functions, instead of being concentrated in the `logical` and `set_cc` functions. Higher-order functions, although they help write concise descriptions and they reduce opportunities for error, can make descriptions unreadable even by domain experts.

Higher-order functions do *not*, however, compromise our ability to analyze λ-RTL descriptions at compile time. Because λ-RTL is based on the *typed* λ-calculus, and because it does not include recursion, the λ-RTL compiler can and does apply all functions at compile time and convert the results to normal form. The type system guarantees that an RTL in normal form contains no λ-abstractions.

A reasonable alternative to λ-RTL's current design would be to eliminate higher-order functions, or even all functions. RTLs would have to be expressed using only RTL primitives. This alternative would give up brevity in favor of a better match between the language and the problem. The benefit of making the descriptions more readable might exceed the cost of making them harder to write. As a compromise solution, we have given the λ-RTL compiler the ability to emit its normal form in TeX, which can be examined to see if the RTLs correspond to the author's intent. Hypertext links within the TeX make it possible to browse the results of simple compiler analyses, e.g., to ask questions like "which instructions perform an integer multiply operation?" Other designers have made similar choices. ISP has first-order functions, and *spawn* has first-class functions. In both languages, functions help make descriptions concise.

Use defaults effectively

Good defaults can make descriptions more concise and readable. They can also reduce opportunities for errors. Domain-specific languages offer especially good opportunities to use defaults, because it may be possible to use knowledge of the domain to infer information that is not stated explicitly. For example, SLED's design exploits two facts about binary representations.

- The binary representation of a typical instruction is determined by inserting the values of its operands into the partial representation associated with the opcode.
- In most RISC instruction sets, the whole instruction fits in a single SLED token (often a machine word).

SLED exploits the first fact by using the same names for fields and operands. The declarations of fields identify the positions they occupy, and by default SLED uses the constraint "`f = f`"[4] to put operands named after fields in the same positions as those fields. SLED exploits the second fact by making the default binary representation be the conjunction of the constraints associated with all operands and the opcode. These defaults are very effective in describing RISC machines. For example, on the SPARC, only the branch, call, and `sethi` instructions require their binary representations to be specified explicitly; the binary representations of all other instructions can be defaulted.

SLED also uses defaults to mitigate the problems of having a single language specify both assembly-language and binary representations. One can specify binary representation without worrying about assembly language, because assembly language comes "for free;" every specification that associates a binary representation with a constructor, explicitly or implicitly, also associates an assembly-language representation with that constructor.

[4]The `f` to the left of the `=` sign is the name of a field; the `f` to the right is the name of an operand.

One cannot quite specify assembly language without worrying about binary representation, because the default binary representation might be implausible, or even inconsistent. One can, however, replace the `constructors` keyword with the `assembly syntax` keyword, in which case only the assembly part of the specification is used. In practice, the ability to specify assembly syntax without binary representation is useful primarily for testing (Fernández and Ramsey 1997).

λ-RTL's use of defaults is less obvious but more pervasive; the reason λ-RTL exists at all is to make it possible to omit detail from machine descriptions. Splitting instructions' semantics among different named attributes makes it easier to specify default behaviors. For example, because our λ-RTL specifications use a separate `trap` attribute to say whether and how an instruction may fault, it is easy to define a default trapping behavior, i.e., never to trap. The regular advance of the program counter is another default, which need not be specified with every instruction.

The most significant use of defaults within λ-RTL is built into λ-RTL's type system. λ-RTL uses type inference with subtyping (Nordlander 1999) not only to infer the types of functions and results, but also to allow much other information to be left implicit.

- In the target RTLs, every operator must be parameterized with its size, so for example, the machine description can distinguish a 32-bit add from a 64-bit add. An extension to Milner's (1978) type-inference algorithm makes it possible for these sizes to be defaulted.
- As noted in the sketch of λ-RTL, we decided not to allow sign-extension or zero-extension operations to be defaulted; operations that change the sizes of values must be given explicitly. In this instance, providing a default operation would not have been effective; it is more important for the λ-RTL translator to detect the ambiguity and require an explicit resolution.
- In the target RTLs, cells must be explicitly aggregated into locations, and locations must be fetched from to produce values. λ-RTL uses subtyping (implicit coercions) to make these operations implicit in the source language; placing a location where a value is expected results in a fetch by default. To know which byte order to use in aggregations, λ-RTL attaches a default byte order to each storage space.

  Typical imperative languages leave fetches implicit,[5] but they do it with *syntax.* Certain locations within the grammar are designated as *lvalue contexts*, others as *rvalue contexts*, and fetches are inserted only

[5]Bliss is a rare exception.

when locations appear in rvalue contexts. We foresaw two problems with using a syntactic approach to implicit coercions in λ-RTL. First, as discussed below, we did not feel we had sufficient grasp of the problem to come up with a good syntax on the first try, and we did not want to freeze a syntax prematurely. Second, we had not two but three contexts to deal with (values, locations, and cells). While it was not clear that there would be a natural syntactic solution to implicit coercion with three contexts, a solution based on subtyping can handle any number of contexts.

- Implicit aggregation requires particular care because not only the aggregation operation itself, but also the size of the result, are left implicit. We would like a notation such as `$r[0] := $m[sp]` to mean something sensible by default, i.e., "starting at the memory address stored in the stack pointer, aggregate 4 8-bit bytes into a 32-bit location, fetch a 32-bit value therefrom, and store the result in register 0." With different defaults, the same notation could also be used to denote a 64-bit load, but in λ-RTL that larger value requires an explicit aggregation.

```
$r[0] := RTL.AGGB #8 #64 $m[sp]
```

It is possible to specify exactly the same effect simply by using a type cast on the right-hand side.

```
$r[0] := ($m[sp] : #64 bits)
```

In either case, on the left-hand side, the aggregation of the register pair (`$r[0]`,`$r[1]`) into a 64-bit location is still implicit.

We have taken some care in formulating λ-RTL's rule for default aggregations. A phrase like l := r, where l is a 32-bit cell and r is an 8-bit cell, results in an aggregation to size n, where n is constrained to be divisible by both 8 and 32. Any collection of such divisibility constraints has a unique least solution, i.e., the least common multiple of all the constants, which could be taken as the default value of n. At a suggestion from Mark Bailey, however, λ-RTL chooses a default n only under more restricted conditions. If one of the constants is a multiple of all the others, that constant is taken as the value of n; otherwise, the λ-RTL program is rejected, and the aggregation must be written explicitly. For example, if l refers to a 36-bit register, and r to an 8-bit byte, λ-RTL does not aggregate together 9 bytes and store the result in a register pair; instead it issues an error message to the effect that because 36 is not a multiple of 8, an explicit aggregation is required.

Don't leave an "escape hatch"

Many domain-specific languages allow programmers to include fragments of code written in a general-purpose programming language, often C. This sort of "escape hatch" provides tremendous expressive power, which can be used to solve the "last 10%" of a problem when the domain-specific language does 90% of the job. In SLED, we were tempted to use just such a technique to manage the encoding of PC-relative branches and calls. Given our goals, however, such a technique has several drawbacks.

- It establishes a particular implementation language, making it less likely that the domain-specific language's compiler will support more than one implementation language for software tools. To support a new implementation language, descriptions must be cloned and modified; they cannot be reused.
- Arbitrary C code is likely to establish a preferred direction, and therefore to preclude some uses of the description. If C code is provided for both directions, there is in general no way to assure that the code given for the two directions is consistent.
- The presence of arbitrary C code reduces the domain-specific compiler's ability to detect errors. Some kinds of errors can be detected only by the C compiler, or perhaps not even then, and error messages issued by the C compiler may be hard to trace to faults in the description.

Adhering to this principle can be expensive. In SLED, for example, handling the difficult encoding cases without arbitrary C code required an equation solver. This solver and its special operators (bit insertion, bit extraction, and sign extension), represent a significant fraction of the effort required to implement SLED. Given other goals, this effort might not have been justified—in many circumstances, 90% solutions with escape hatches are a good design technique, especially when a preferred direction and preferred implementation language have been established.

Eliminating escape hatches can also be a barrier to a different kind of reuse—not the reuse of descriptions, but the adaptation of descriptions and tools to new purposes. If a language has an escape hatch, a user may well be able to adapt a description to a new purpose simply by changing the general-purpose code. For example, vpo's machine descriptions associate arbitrary C code with each instruction's description (Benitez 1994). While this C code is commonly used to check semantic constraints (e.g., two operands are identical, a constant is within a required range, etc.), the escape hatch has allowed vpo to be adapted to new domains. For example, an architect can use vpo to evaluate a potential extension to an instruction set (Davidson and Whalley 1991). The architect adds a new instruction to vpo's machine description, and the C code emits existing instructions that simulate the effect of the new instruction. The escape hatch may also be used to add instrumentation, e.g., to gather address traces for research in memory systems (Whalley 1993) or to help estimate the worst-case execution time of a real-time system (Harmon, Baker, and Whalley 1992). If were no escape hatch, adapting machine descriptions to such new uses might require understanding and adapting the language's supporting tools, or even writing new tools—a much larger effort.

Make syntax resemble informal notations used in the domain

Many problem domains have established notational conventions. Language designers should exploit these conventions to make their languages easier for domain experts to read and write.

SLED applies this principle only in part. As shown above, patterns defined using generating expressions can be made to resemble the opcode tables found in architecture manuals. This resemblance makes it very easy to inspect those parts of a SLED description. SLED descriptions of assembly-language syntax deliberately resemble "suggested assembly language syntax" as shown in manuals. As in manuals, concatenation is implicit, and strings representing punctuation (commas, brackets, stars, etc.) need not be quoted. Even whitespace between operands is significant, since it is usually desired, e.g., in the outputs of disassemblers. Adherence to this principle came at some cost in implementation complexity; the lexer-parser interactions that govern the significance of white space are not for the faint of heart.

Because informal notations are often heavily overloaded, using informal notations can make a language more difficult to design and more difficult to understand. For example, machine designers refer informally to locations without making careful distinctions about how big the locations are or whether they are really referring to the values contained in the locations. In λ-RTL, we use a sophisticated type system to determine whether the expression `$m[sp+12]` stands for a byte in memory, a word in memory, or perhaps for the value stored in that byte or that word. As another example, informal descriptions of machine instructions seldom have to make the distinction between a field in the binary representation named `rs1` and an operand named `rs1`. Because this distinction is necessary in SLED, a reader may sometimes have to cope with such inscrutable phrases as `rs1 = rs1` or `disp30 = disp30`.

There are two ways in which SLED departs from existing notations: in the assignment of positions to fields, and in the concatenation of multiple instruction tokens to form CISC instructions. Domain experts don't use textual notations to talk about these aspects of binary representation; they draw pictures. We felt there were compelling advantages to making the SLED language simple ASCII, even though ASCII is not the notation commonly used in the domain. For example, tools for editing ASCII representations are universally available, and it is easy to write new tools to both analyze and generate ASCII. Nevertheless, for students in particular, a "visual" front end for SLED might be very useful.

The earlier principle of using separate languages for separate properties can make it easier to use notations common in a domain. For example, machine pipelines and datapaths are also commonly described using pictures. Milner (1999) proposes a visual language for describing processor pipelines; despite having a dramatically different syntax and a structure not based on constructors, this language will interoperate with λ-RTL and SLED, and it should simplify the creation of instruction schedulers. It should be considered a member of our confederation.

For λ-RTL, it unclear *which* of the many notations for register transfers should be enshrined in the language. Should assignment be notated `<-`, `:=`, or `=`? Both parallel and sequential composition are useful; which one should the semicolon denote? λ-RTL does not prefer any of these notations over any of the others, which brings us to another principle.

When you aren't sure what to do, let the users decide

During the development of λ-RTL, it wasn't clear to us what notations users would find most readable for describing RTLs, or what language constructs would best help them deal with the repetition involved in specifying similar instructions. Rather than build into λ-RTL our tentative solutions to these problems, we built in two mechanisms that we hoped would make λ-RTL flexible enough so that our users could discover better solutions.

- λ-RTL provides infix operators with arbitrary precedence and associativity. Precedence is assigned using floating-point literals with arbitrary precision, so it is always possible to insert a new operator with precedence between that of any two existing operators. For example, if for some reason an author wanted to use C conventions for infix operators, he or she could introduce these operators, with all 14 levels of precedence, anywhere in the precedence hierarchy.

Infix notation also makes it possible to define "mixfix" operations by using various infix operations with prefix function application. For example, the description of the Pentium's logical instructions can be rewritten to use infix notation instead of `llr`.

```
default attribute of
  [AND OR XOR]^mr^[b ow od] (addr, reg) is
    logical binary
    (addr <== addr <[and or xor]> R reg) [b w d]
```

- Higher-order functions allow λ-RTL users to define their own notations for computations that would otherwise require new language features. Ramsey and Davidson (1998) defines a `do8` function that can repeat the same action on groups of 8 registers; it is helpful in describing the SPARC's register windows.

This flexibility gives users tools they can use to make descriptions more readable and more concise, but it comes at a cost. λ-RTL users must have a deep understanding of both higher-order functions and "stupid infix tricks" in order to exploit the flexibility. Generating expressions create even more flexibility. All this flexibility can be used to create ugly, unreadable descriptions just as easily, if not more easily, than to create good ones. We hope this risk is justified because it avoids the costs of settling prematurely on an inferior notation. Crucially, we do not *require* anyone to use these sharp tools. Not knowing the best way to notate the descriptions, we let the users decide.

Remember a good thief is worth ten scholars

The primary intellectual effort in SLED and λ-RTL has been in the development of the underlying semantic models and abstractions, and in the development of the compile-time analyses, not in the design of the language features. Much of λ-RTL was stolen directly from ML; the most unusual part of SLED was stolen from Icon. Hoare (1989) expresses this principle somewhat differently.

> The language designer should be familiar with many alternative features designed by others, and should have excellent judgement in choosing the best and rejecting any that are mutually inconsistent... One thing he should not do is to include untried ideas of his own. His task is consolidation, not innovation.

Describe at least three machines

Chris Fraser encouraged us never to consider a machine-description framework finished until it had described at least three machines—and one of those had to be hard. For many purposes, the Pentium can be relied on to be hard. Describing the Pentium required deep changes in SLED's predecessor, and it's not clear how languages like nML, LISAS, or *spawn* would extend gracefully to the Pentium. In λ-RTL, describing the Pentium's context-dependent addressing modes was a challenge, and so was the description of the SPARC's register windows.

In our experience, three machines are enough to work most of the bugs out of a language design, but not all. For example, it wasn't until machine number five or six (the PowerPC) that we realized SLED should support two bit numberings (bit 0 as either the most significant or the least significant bit in a token).

Use an appropriate language for each sub-domain
Avoid a preferred direction
Analyze at compile time
Help manage repetition
Make abstractions match the problem
Use defaults effectively
Don't leave an "escape hatch"
Make syntax resemble standard notations
When you aren't sure, let the users decide
Remember a good thief is worth ten scholars
Describe at least three machines
Design for testability

Machine descriptions should be *reusable*, *partial*, *reliable*, *easy to read and write*, and *concise*.

Table 1: Design principles and goals

Design for testability

Testing a machine description against some independent standard is an invaluable mechanism for increasing confidence in the reliability of the description. It should appear higher on this list, but we were unable to apply this principle to either SLED or λ-RTL. In the case of SLED, we designed to other criteria, then figured out how to test after the fact (Fernández and Ramsey 1997). In the case of λ-RTL, it is still unclear how best to test the descriptions.

6 Summary

Table 1 lists the principles and goals set forth in this paper. To summarize, we review the connections between principles and goals.

- To make descriptions *reusable*, we may use different languages for different sub-domains, we avoid establishing a preferred direction, and we don't allow "escapes" to arbitrary code.

- To enable *partial* descriptions, we use different languages. λ-RTL's attributes also make it easier to write useful partial descriptions.

- To help users create *reliable* descriptions, we analyze them at compile time, help manage repetition, use abstractions that match the problem, use defaults effectively, and use syntax that resembles standard notations. When possible, we also test descriptions, although we have not succeeded in "designing in" features for testability. Making descriptions concise and readable also contributes to reliability.

- To make descriptions *easy to read and write*, we use languages designed for each sub-domain, we use abstractions that match the problem, and we use syntax that resembles standard notations used in the problem domain. When we're not sure of a good notation, we give users the power to define their own, e.g., through the use of higher-order functions and infix operators of arbitrary precedence.

- To make descriptions *concise*, we use languages designed for each sub-domain, we help manage repetition, and we use defaults effectively.

Finally, we have followed Tony Hoare's general advice to use other people's ideas, and we have followed Chris Fraser's specific advice not to consider a machine-description language finished until it has described at least three machines.

Acknowledgments

We stole the phrase "a good thief is worth ten scholars" from Randy Pausch.

Paul Hudak provided helpful comments on a draft of the manuscript.

This work has been supported in part by National Science Foundation grants ASC-9612756 and CCR-9733974 and by DARPA contract MDA904-97-C-0247.

References

Aigrain, Philippe, Susan L. Graham, Robert R. Henry, Marshall Kirk McKusick, and Eduardo Pelegrí-Llopart. 1984. Experience with a Graham-Glanville style code generator. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 13–24. ACM, ACM.

Arvind and Xiaowei Shen. 1999 (May/June). Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3). Special Issue on Modeling and Validation of Microprocessors.

Augustsson, Lennart. 1985 (September). Compiling pattern matching. In Jouannaud, Jean-Pierre, editor, *Functional Programming Languages and Computer Architecture*, LNCS.

Bailey, Mark W. and Jack W. Davidson. 1995 (January). A formal model and specification language for procedure calling conventions. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 298–310, San Francisco, CA.

———. 1996a (February). Reusable application-dependent machine descriptions. Presented at WCSSS'96. May be available from `http://www.cs.virginia.edu/~mwb5y/`.

———. 1996b (May). Target-sensitive construction of diagnostic programs for procedure calling sequence generators. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 31(5):249–257.

Bala, Vasanth, Evelyn Duesterwald, and Sanjeev Banerjia. 2000 (May). Dynamo: A transparent dynamic optimization system. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 35(5):1–12.

Ball, Thomas and James R. Larus. 1994 (July). Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360.

Barbacci, Mario R. 1975 (February). A comparison of register transfer languages for describing computers and digital systems. *IEEE Transactions on Computing*, C-24:137–150.

Barbacci, Mario R. and Daniel P. Siewiorek. 1977 (October). Evaluation of the CFA test programs via formal computer descriptions. *Computer*, 10(10):36–43.

———. 1982. *The Design and Analysis of Instruction Set Processors.* New York, NY: McGraw-Hill.

Baudinet, Marianne and David MacQueen. 1985 (December). Tree pattern matching for ML (extended abstract). Unpublished manuscript, AT&T Bell Laboratories.

Bell, C. Gordon and Allen Newell. 1971. *Computer Structures: Readings and Examples.* New York: McGraw-Hill.

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

———. 1994 (March). The advantages of machine-dependent global optimization. In Gutknecht, Jürg, editor, *Programming Languages and System Architectures*, Vol. 782 of *Lecture Notes in Computer Science*, pages 105–124. Springer Verlag.

Benitez, Manuel Enrique. 1994 (May). *Register Allocation and Phase Interactions in Retargetable Optimizing Compilers*. PhD thesis, University of Virginia, Department of Computer Science.

Bradlee, David, Robert Henry, and Susan Eggers. 1991 (June). The Marion system for retargetable instruction scheduling. *ACM SIGPLAN '91 Conference on Programming Language Design and Implementaion,* in *SIGPLAN Notices*, 26(6):229–240.

Cardelli, Luca. 1984 (August). Compiling a functional language. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 208–217. ACM, ACM.

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

Cmelik, Bob and David Keppel. 1994 (May). Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137.

Cook, Todd and Ed Harcourt. 1994 (May). A functional specification language for instruction set architectures. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 11–19.

Davidson, Jack W. 1981. *Simplifying Code Generation Through Peephole Optimization*. PhD thesis, Dept. of Computer Science, University of Arizona, Tucson, AZ.

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

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

Davidson, Jack W. and David B. Whalley. 1991 (November). A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems*, 15(9):459–472.

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

Fauth, Andreas, Johan Van Praet, and Markus Freericks. 1995 (March). Describing instruction set processors using nML. In *The European Design and Test Conference*, pages 503–507.

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

Fernández, Mary F. 1995a (November). *A Retargetable Optimizing Linker*. PhD thesis, Dept of Computer Science, Princeton University.

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

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

Graham, Susan L., Robert R. Henry, and Robert A. Schulman. 1982. An experiment in table driven code generation. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 32–43. ACM, ACM.

Griswold, Ralph E. and Madge T. Griswold. 1996. *The Icon Programming Language*. Third edition. San Jose, CA: Peer-to-Peer Communications.

Hadjiyiannis, George, Silvina Hanono, and Srinivas Devadas. 1997. ISDL: An instruction set description language for retargetability. In *Proceedings of 34th Design Automation Conference*, pages 299–302, New York, NY.

Hadjiyiannis, George. 1998. *ISDL: Instruction Set Description Language, User's Manual*. Computer-Aided Automation Group, MIT. See `http://caa.lcs.mit.edu/caa/publications.html`.

Harmon, Marion G., T. P. Baker, and David B. Whalley. 1992 (December). A retargetable technique for predicting execution time. In Werner, Robert, editor, *Proceedings of the Real-Time Systems Symposium*, pages 68–77, Phoenix, AZ.

Hastings, Reed and Bob Joyce. 1992 (January). Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, San Francisco, CA.

Hoare, C. A. R. 1989. Hints on programming-language design. In Jones, C. B., editor, *Essays in Computing Science*, Chapter 13, pages 193–216. New York, NY: Prentice Hall. Adapted from Hoare's keynote address to the first Symposium on Principles of Programming Languages.

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

Hughes, R. John M. 1989 (April). Why functional programming matters. *The Computer Journal*, 32(2):98–107.

Larus, James R. and Eric Schnarr. 1995 (June). EEL: machine-independent executable editing. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 30(6):291–300.

Laville, Alain. 1991 (April). Comparison of priority rules in pattern matching and term rewriting. *Journal of Symbolic Computation*, 11(4):321–347.

Lipsett, Roger, Carl F. Schaefer, and Cary Ussery. 1989. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers.

Maranget, Luc. 1992 (June). Compiling lazy pattern matching. In White, Jon L., editor, *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 21–31, San Francisco, CA.

Michael, Neophytos G. and Andrew W. Appel. 2000 (January). Machine instruction syntax and semantics in higher order logic. Unpublished manuscript, Princeton University.

Milner, Christopher W. 1999 (March). Pipeline descriptions for retargetable compilers: A decoupled approach. Technical Report CS-99-11, Department of Computer Science, University of Virginia.

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

Milner, Robin, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised)*. Cambridge, Massachusetts: MIT Press.

Morrisett, Greg, David Walker, Karl Crary, and Neal Glew. 1998 (January). From System F to typed assembly language. In *Conference Record of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, California.

Necula, George C. 1997 (January). Proof-carrying code. In *Conference Record of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France.

Nordlander, Johan. 1999 (January). Pragmatic subtyping in polymorphic languages. *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming,* in *SIGPLAN Notices*, 34(1):216–227.

Oakley, J. D. 1979 (April). Symbolic execution of formal machine descriptions. Technical Report CMU-CS-79-117, Computer Science Department, Carnegie-Mellon University.

Peyton Jones, Simon L., John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. 1999 (February). Haskell 98: A non-strict, purely functional language. Available from `www.haskell.org`.

Peyton Jones, Simon L., Erik Meijer, and Daan Leijen. 1998 (June). Scripting COM components in Haskell. In *Fifth International Conference on Software Reuse*, pages 224–233, Los Alamitos, CA.

Proebsting, Todd A. and Christopher W. Fraser. 1994 (January). Detecting pipeline structural hazards quickly. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 280–286, Portland, OR.

Ramsey, Norman. 1996 (April). A simple solver for linear equations containing nonlinear operators. *Software—Practice & Experience*, 26(4):467–487.

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

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

Ramsey, Norman and David R. Hanson. 1992 (July). A retargetable debugger. *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 27(7):22–31.

Romer, Ted, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and J. Bradley Chen. 1997 (August). Instrumentation and optimization of Win32/Intel executables using Etch. In *USENIX Windows NT Workshop*, pages 1–7, Berkeley, CA, USA.

Rosenblum, Mendel, Edouard Bugnion, Scott Devine, and Stephen Alan Herrod. 1997 (January). Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103.

Schnarr, Eric and James R. Larus. 1996 (December). Instruction scheduling and executable editing. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 288–297, Paris, France.

Sites, Richard L., Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. 1993 (February). Binary translation. *Communications of the ACM*, 36(2):69–81.

SPARC International. 1992. *The SPARC Architecture Manual, Version 8*. Englewood Cliffs, NJ: Prentice Hall.

Srivastava, Amitabh and David W. Wall. 1993 (March). A practical system for intermodule code optimization. *Journal of Programming Languages*, 1:1–18. Also available as WRL Research Report 92/6, December 1992.

Stallman, Richard M. 1992 (February). *Using and Porting GNU CC (Version 2.0)*. Free Software Foundation.

Thomas, Donald and Philip Moorby. 1995. *The Verilog Hardware Description Language*. 2nd edition. Norwell, USA: Kluwer Academic Publishers.

Wahbe, Robert, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993 (December). Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 203–216.

Whalley, David B. 1993 (January). Techniques for fast instruction cache performance evaluation. *Software—Practice & Experience*, 23(1):95–118.

Wick, John Dryer. 1975 (December). *Automatic Generation of Assemblers*. PhD thesis, Yale University.