Pragmatic Aspects of Reusable Program Generators

Norman Ramsey
Division of Engineering and Applied Sciences
Harvard University
(*e-mail:* `nr@eecs.harvard.edu`)

Abstract

When writing a program generator requires considerable intellectual effort, it is valuable to amortize that effort by using the generator to build more than one application. When a program generator serves multiple clients, however, the implementor must address pragmatic questions that implementors of single-use program generators can ignore. In how many languages should generated code be written? How should code be packaged? What should the interfaces to the client code look like? How should a user control variations? This paper elaborates on these questions by means of case studies of the New Jersey Machine-Code Toolkit, the λ-RTL Translator, and the ASDL program generator. It is hoped that the paper will stimulate the development of better techniques. Most urgently needed are a standard way to support multiple target languages and a simple, clear way to control interfaces to generated code.

1 Introduction

There are many ways to deploy program generators; not all are alike. For example, some generators may be used to build only one application, while others may be used in many different contexts. This paper examines program generators that are intended to be reused—what properties they should have and how they might be structured.

1.1 Why reuse?

Not all program generators need to be reusable. Many are simple tools that are unknown to all but their authors; writing programs that generate programs is a technique every experienced programmer should use from time to time (Hunt and Thomas 1999, Section 20). Even program generators that become known to many programmers are not necessarily reusable. The `lburg` tool used to generate `lcc`'s code generators (Fraser and Hanson 1995), the *spawn* tool used to generate executable editors (Larus and Schnarr 1995), and the tool that `gcc` uses to compile its machine descriptions (Stallman 1992) are all examples. Someone building a new code generator, a new executable editor, or a new peephole optimizer[1] would have to create a new program generator.

A few program generators are intended for reuse. Examples include the lexer generator Lex (Lesk and Schmidt 1975), the parser generator Yacc (Johnson 1975), and the code-generator generator BURG (Fraser, Henry, and Proebsting 1992). The effort required to create a reusable program generator is much greater than that required to create a single-use program generator, so it is worth examining what benefits might justify such an effort.

- Generating a lexical analyzer requires algorithms that manipulate finite automata. Such an algorithm may translate a regular expression to an NFA, convert an NFA to a DFA, or minimize the number of states in a DFA. These algorithms are worth reusing. Perhaps more importantly, the lexical analyzer is a significant performance bottleneck in many compilers (Waite 1986), and it is worth trying to automate the expertise needed to make this component efficient. Newer lexer generators attempt to provide not only good automata but also efficient code (Gray 1988; Bumbulis and Cowan 1993).
- Generating a parser requires analysis of the grammar and construction of lookahead sets. Writing an LR parser and building SLR(1) parsing tables is fairly straightforward (Aho and Johnson 1974), but building the more useful LALR(1) tables is harder, especially if one wishes to do so efficiently (DeRemer and Pennello 1982).
- Some code generators select instructions by finding minimum-cost covers of intermediate-code trees. A typical code generator might use dynamic programming to find a minimum-cost cover, and it is fairly easy to write a program to generate such a code generator. But to squeeze the last ounce of performance out of the code generator, getting the fastest possible compiler, it is desirable to do the dynamic programming at program-generation time. Doing so efficiently is not so easy (Proebsting 1992).

In each of these examples, the program generator is useful because it encodes a difficult algorithm that can be used again and again, because it solves a performance problem that occurs again and again, or both.

These and other examples show that if a nontrivial algorithm or technique is required to create efficient code, that algorithm or technique may be worth embodying in a program generator. *How* to do so is the central, domain-specific question of any particular program-generation problem. This paper discusses a different, pragmatic question: how can we arrange for generated code to work well with the client code? A reusable program generator should generate code that can fit into a variety of applications, written in a variety of implementation languages.

When a program generator is not reusable, we wind up with variations. For example, iburg, lburg, nburg, ML-Burg, and jburg are all variations on BURG; the

[1] `Gcc`'s machine descriptions contain all manner of information used to generate the compiler, but the core of the system is based on instruction selection by peephole optimization (Davidson and Fraser 1984).

different variations support either different implementation languages or different ways of specifying the connections between the tiles that BURG uses to cover a tree and the actions that are to be taken once a minimum-cost cover is found. In the process of providing this additional support, one of the significant benefits of Proebsting's original program generator has been lost: not one of the variations does dynamic programming at program-generation time. While users might disagree over the value of doing the dynamic programming at program-generation time, many users do not have the option because Proebsting's original program generator is not reusable.

The primary contribution of this paper is to present problems that make it difficult to build reusable program generators. The paper reports on experience with the program generators behind the New Jersey Machine-Code Toolkit (Ramsey and Fernández 1995), the λ-RTL Translator (Ramsey and Davidson 1999), and the Abstract Syntax Description Language (Wang et al. 1997). A study of only three tools cannot be comprehensive, and studying more tools, such as the GenVoca line of program generators (Batory et al. 2000), the Horus stub generator (Gibbons 1987), or the Mockingbird stub compiler (Auerbach et al. 1999), might yield more insight. Still, the three tools in this study span a range of interseting program-generation tasks, and from these tools we can learn much about the problems of making program generators reusable.

The pragmatic aspects of program generation cut across the domain-specific aspects. If the pragmatic problems can be solved, therefore, the solutions might well apply to many domain-specific program generators, each of which could be made reusable. If solutions can be realized in a reusable infrastructure like GenVoca, we will take a big step forward in our ability to create reusable program generators. A second contribution of this paper, therefore, is to discuss pragmatic aspects of the implementation techniques used in the three tools of our study. The discussion focuses on intermediate languages, which can represent both code to be generated and the precursors of such code. The discussion identifies the elements of these languages that might be valuable in a reusable infrastructure.

1.2 Requirements for reuse

If you want to use a program generator, what should you demand?

- The generated code should be *written in the programming language of your choice.* Foreign-function calls may suffice in some cases, but for most purposes, the generated code needs to work in the client's native language. This requirement explains the repeated pleas for "Yacc-alikes" on the Usenet newsgroup `comp.compilers`.
- The generated code should be *idiomatic and readable.* Because Lex and Yacc have been used for a long time, you might be willing to trust their output without reading it, but code generated by an unproven generator should be scrutinized. Wise programmers distrust generated code that they don't understand (Hunt and Thomas 1999, Section 35).

Prettyprinting code nicely is not enough; the code must use programming conventions that are native to the target language. Such conventions encompass decisions ranging from the spelling of identifiers (use of capital letters and underscores) to the representations of common data structures (lists, tables) to input/output conventions.
- The program generator should *detect errors in specification*. It should not generate incorrect code that fails at a later stage, e.g., that fails to compile.
- The generated code should offer *acceptable performance*.
- You should *control the interfaces* between generated code and client code. In many modern programming languages, even simple procedural interfaces offer considerable scope for variation. Should a C procedure be called directly or through a pointer? Should a Modula-3 procedure be called through a procedure appearing in an interface or through a method appearing in an object type? Should C++ procedures be generated at top level, in their own class, or in their own namespace?
  These choices are not always matters of personal preference. For example, if the client program will contain multiple instances of an abstraction (some of which may be generated automatically), the indirect call, the object type, or the class are the better alternatives. But if there is to be only one instance of the abstraction, and if performance is critical, the cost of indirection may be too high. These tradeoffs are illustrated below with examples from the New Jersey Machine-Code Toolkit.
- You may wish to *control internals* of the generated code, for example, to adjust space/time tradeoffs or other properties that affect performance.

Two of these requirements, error detection and performance, seldom affect reusability. The other requirements affect reusability insofar as they are likely to change. Most obviously, different clients may require different implementation languages, different interfaces, and different internals. If you are writing your own program generator, of which you intend to write the only client, the right language, interfaces, and internals should emerge naturally. It is much harder to create a program generator intended for reuse; not all clients have the same needs, and after all, these properties appear peripheral to a program generator's main job, which is to embody some useful, domain-specific algorithm or technique. The primary claim of this paper is that these pragmatic aspects of program generation are every bit as important as the domain-specific, algorithmic aspects. This claim is supported by experience with the New Jersey Machine-Code Toolkit, the λ-RTL Translator, and the Abstract Syntax Description Language program generator.

This paper presents experience in the form of case studies. Each case study begins with an overview, which describes the sort of client in which the generated code is embedded. The overview also explains what parts of the client's problem are solved by the program generator. The rest of the case study answers the following questions:

1. What does the generated code do?
2. What are the interfaces between generated code and client code?

3. What controls the output of the program generator—both the interfaces and the internals?
4. How and how well does the tool meet the requirements set forth above?

2 Case Study: The New Jersey Machine-Code Toolkit and λ-RTL Translator

The New Jersey Machine-Code Toolkit (Ramsey and Fernández 1995) is a program generator that helps programmers create applications that work with *representations* of machine instructions, especially the binary representation. Such applications include assemblers, disassemblers, code generators, tracers, profilers, debuggers, and executable editors, for example. The code generated by the Toolkit handles the low-level bit fiddling, which is tedious and error-prone to do by hand. Among other applications, the Toolkit has been used to build a retargetable debugger (Ramsey and Hanson 1992), a retargetable, optimizing linker (Fernández 1995), a run-time code generator, a decompiler, an execution-time analyzer (Braun 1996), an optimizing compiler for object-oriented languages (Dean et al. 1996), and a binary translator (Cifuentes, van Emmerik, and Ramsey 1999).

The λ-RTL Translator (Ramsey and Davidson 1998) is a companion to the Toolkit. It is intended to help programmers create applications that work with the *semantics* of machine instructions. While much less mature than the Toolkit, the λ-RTL Translator has been used in compiler construction and to build tools that analyze binary code. The two program generators are similar in concept, and their implementations share code, so it makes sense to consider them together.

In discussing the Toolkit, I use first-person plural to describe joint work with Mary Fernández, the Toolkit's co-creator.

2.1 What the generated code does

The purpose of the code generated by the Toolkit and the λ-RTL Translator is to map into and out of different representations of machine instructions. Between them, the tools support five representations.

- *Binary code* is the hardware's representation of instructions. For example, on the SPARC, the instruction that takes the doubleword floating-point quotient of values in registers 10 and 12, putting the result in register 14, has the binary representation depicted here:

2	14	52	10	78	12
31 30	29 25	24 19	18 14	13 5	4 0

  Because not every instruction on every machine fits in a machine word, the Toolkit cannot simply use machine words to represent binary code. Instead, the Toolkit requires that binary code be part of an *instruction stream*. An instruction stream is like a byte stream, except that the units, which are called "tokens," need not be 8-bit bytes; the Toolkit permits tokens to have any size. Within a stream, an instruction is a sequence of one or more tokens;

for example, the SPARC instruction shown above is one 32-bit token, but a Pentium instruction might include several 8-bit prefixes, an 8-bit opcode, 8-bit format bytes, and a 32-bit immediate operand, each of which would be considered a token.

To use code generated by the Toolkit, a client must provide an implementation of the instruction-stream abstraction. Instruction streams can take many forms; for example, a debugger can treat the text segment of a target process as an instruction stream.

- *Assembly language* is a textual representation of instructions. Assembly language is typically supported by assemblers, disassemblers, compilers, and debuggers. For example, on the SPARC, the doubleword floating-point divide instruction shown above has the following representation in the assembly language suggested by the standard (SPARC 1992):

```
fdivd %f10, %f12, %f14
```

  Many machines have a single, standard assembly language, but for popular platforms such as the Pentium, it is common for different software families to use different assembly languages.
- The Toolkit generates an *algebraic data type* that represents instructions. This data type stands in the same relation to an assembly language that an abstract-syntax tree stands in relation to a programming language. In the case of the Toolkit, it not only abstracts away from the details of concrete syntax but also gives each instruction a unique name, eliminating any overloading. Overloading the names of instructions is common in assembly languages.

  For example, here are excerpts from the data types corresponding to the SPARC instruction set, as generated in the Objective Caml language:

```
type address = ...

type t
  = Ldsb of address * nativeint
  | Ldsh of address * nativeint
  ...
  | Fmuld of nativeint * nativeint * nativeint
  | Fdivd of nativeint * nativeint * nativeint
  | Faddq of nativeint * nativeint * nativeint
  ...
```

- *Register transfers* describe the computational effects of instructions. They have a long been used in research both on compilers and on computer architecture (Davidson 1981; Barbacci and Siewiorek 1982). For example, the computational effect of the floating-point divide instruction may be written using the following register transfers, which are notated in λ-RTL:

```
$f[14] := fdiv ($f[10], $f[12], RD) : #64 bits
```

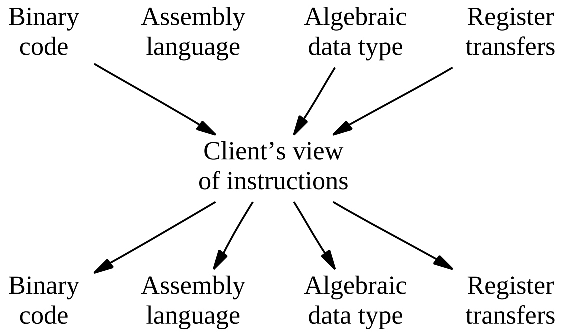

Fig. 1. Mappings implemented by the Machine-Code Toolkit and λ-RTL Translator
Given any path through the graph above, the Toolkit and Translator can cooperate to generate a mapping corresponding to that path. Examples appear in the text.

The identifier `RD` represents the location containing the machine's floating-point rounding modes. The λ-RTL notation above is expanded into a fully elaborated form, as described by Ramsey and Davidson (1998).

The λ-RTL Translator is intended to support different representations of register transfers for use in different clients. Although the Translator currently supports three clients written in two languages (C and Objective Caml), its support for multiple representations of register transfers is still immature. This support is therefore beyond the scope of this paper.

- The *client's view of instructions* is embodied in an interface generated by the Toolkit. This interface can take a number of forms. In each form, there is a function, procedure, or method for each instruction. For example, here is a fragment of an interface to SPARC instructions, written in the Objective Caml language:

```
module type Sparc.S = sig
  type reg_or_imm
  type address
  type regaddr
  type t
    ...
  val ldsb : address:address -> rd:nativeint -> t
    ...
  val ldub : address:address -> rd:nativeint -> t
  val fdivd : fs1:nativeint -> fs2:nativeint -> fd:nativeint -> t
    ...
end
```

Now we can answer the question "what does the generated code do?" The Toolkit and Translator generate programs that implement mappings between the five representations listed above. In addition, the Toolkit can generate very flexible recognizers for binary code.

Figure 1 shows a pictorial view of the mappings. The mappings at the top, called *decoders*, map binary code, the data type, or register transfers to instructions. The Toolkit cannot currently generate parsers, which would be needed to map assembly language to instructions, so that arrow is missing in Figure 1. The mappings at the bottom of Figure 1, called *encoders*, map from instructions to binary, assembly language, the algebraic data type, or register transfers. The Toolkit and Translator can also cooperate to generate deforested, combined mappings, e.g., directly from binary code to register transfers. Most mappings are done in a single step, but an encoder mapping into binary code may be split into two stages, as described below. The rest of this section illustrates some of the mappings with examples.

A map from a client's view to binary code is implemented by an *encoding procedure*. Here is one written in the C language. The macro `emitm` is used to emit binary code into the "current instruction stream," which is stored in global mutable state.

```
void fdivd(unsigned fs1, unsigned fs2, unsigned fd) {
  emitm(2 << 30 | 52 << 19 | 78 << 5 | fs1 << 14 | fs2 | fd << 25, 4);
}
```

This kind of mapping may be used to generate binary code in an assembler, dynamic compiler, or other program that emits binary; Fernández (1995) describes its use in a whole-program optimizer.

Here is a fragment of the mapping from the client's view to assembly language.

```
static void print_unsigned_fd(unsigned fd) {
  static char *fd_names[] = {
    "%f0", "%f1", "%f2", "%f3", ..., "%f30", "%f31",
  };
  asmprintf(asmprintfd, "%s", fd_names[fd]);
}

void fdivd(unsigned fs1, unsigned fs2, unsigned fd) {
  asmprintf(asmprintfd, "%s", "fdivd ");
  print_unsigned_fd(fs1);
  asmprintf(asmprintfd, "%s", ", ");
  print_unsigned_fd(fs2);
  asmprintf(asmprintfd, "%s", ", ");
  print_unsigned_fd(fd);
  asmprintf(asmprintfd, "\n");
}
```

This kind of mapping would typically be used in a compiler to emit assembly language. It might also be part of a debugger or disassembler.

Here is a mapping from the client's view to register-transfer lists.

```
Rtl_ty_rtl fdivd (unsigned fs1, unsigned fs2, unsigned fd)
{
```

```
  return Rtl_assign(Rtl_location('f', fd), 64,
             Rtl_apply(Rtl_mkOperator(Rtl_op_fdiv, 64),
                       Rtl_fetch(Rtl_location('f', fs1), 64),
                       Rtl_fetch(Rtl_location('f', fs2), 64),
                       Rtl_fetch(Rtl_location('F', Rtl_int(0)), 2),
                       NULL));

}
```

This mapping has been used to help connect the `lcc` compiler (Fraser and Hanson 1995) to the Very Portable Optimizer (Benitez and Davidson 1988).

Mapping into binary code is actually more complicated than is clear from the example above. The problem is that operands of some instructions may be bound to values at two different times: assembly time and link time. For example, in an instruction such as `fdivd %f10, %f12, %f14`, the values of the operands (10, 12, and 14) are known at the time the instruction is emitted, i.e., at assembly time. In an instruction such as `call printf`, however, the value of `printf` is typically not available until link time. In standard linkers, this problem is dealt with by a mechanism called *relocation*. The Toolkit deals with the problem by treating the client-to-binary mapping as a *Curried* function (Ramsey 1996a). A generated encoding procedure may not only emit binary code but also produce a closure. The code part of that closure is called a *relocation procedure*.

The most interesting mapping generated by the Toolkit is the mapping shown by the upper left arrow in Figure 1: a mapping from binary code to the client's view of instructions. Such a mapping amounts to an instruction decoder, which is implemented as a decision tree. In C, such a tree may be represented as a nest of `switch` statements; here, for example, is the leaf that recognizes the `fdivd` instruction:

```
  ...
  switch (MATCH_w_32_0 >> 5 & 0x1ff) {
    ...
    case 78: {
      unsigned fs1 = MATCH_w_32_0 >> 14 & 0x1f;
      unsigned fd  = MATCH_w_32_0 >> 25 & 0x1f;
      unsigned fs2 = MATCH_w_32_0 & 0x1f;
      fdivd(fs1, fs2, fd);
      break;
    }
    ...
  }
  ...
```

The Toolkit's ability to generate *efficient* decoders is highly valued by the Toolkit's users; the next section describes a more flexible interface to this ability.

2.2 Interfaces between client code and generated code

There are interface layers both above and below the code generated by the Toolkit. Above, clients need interfaces to generated encoders and decoders. Below, generated encoders and decoders need interfaces to instruction streams. Also below, code that emits assembly language needs an interface for printing.

Interfaces to instruction streams

Just as many civilized programming languages use two interfaces for I/O streams, the Toolkit uses two interfaces for instruction streams, depending on whether the stream is to be read (in a decoder) or written (in an encoder).

When writing an instruction stream, C code generated by the Toolkit uses a single C procedure or macro as its interface; in the first implementation of `fdivd` on page 8, this procedure is called `emitm`. Each encoding procedure calls `emitm` to emit a token into the instruction stream. At program-generation time, the Toolkit's user can choose a name other than `emitm` by giving a command-line option. The instruction stream that is modified by `emitm` is global mutable state. The Toolkit includes a 600-line library that implements an instruction-stream abstraction and several emission procedures and macros, but application programmers can replace that library with their own code.

When reading an instruction stream, the code generated by the Toolkit uses one type and three phrases as its interface. The type is an *address*; it is abstract and represents a location within an instruction stream. Client code supplies a *representation* of addresses, as well as code phrases that perform three operations:

- Add an integer to an address
- Fetch a token from an address
- Convert an address to an integer

The decoder uses arithmetic and fetching to get at the parts of an instruction, and it may convert addresses to integers, e.g., to extract bits from the program counter.

Interfaces to encoding procedures

Our model of the encoding interface is "something just like assembly language, but more flexible than a traditional assembler." We want to emit instructions by name and to have the assembler make sure the types and values of the operands are sensible. For this reason, the Toolkit associates an encoding procedure with each machine instruction; the procedure can be named after the instruction, and the type system can enforce some useful restrictions on operands. The examples above show interfaces and implementations in which clients call encoding procedures directly, but the Toolkit can also generate interfaces and implementations that support indirect calls.

Independent of direct versus indirect calls, there is another set of variations in a client's view of the *interface* to encoding.

- In the *applicative view*, an instruction is an abstraction, which is built by applying constructors. To get binary code, the client program applies a function to an instruction or to a sequence of instructions. This view is congenial to clients written in applicative style, like the Standard ML of New Jersey compiler.
- In the *imperative view*, there is no abstraction representing instructions, and instructions are not constructed; instead, to each instruction there corresponds a procedure, which is executed for its side effect, e.g., emission of the binary representation of the instruction. This view, which is the one shown in the examples above, is congenial to traditional assemblers.
- The default view supported by the Toolkit is actually a hybrid view. The "effective address"[2] is treated as an abstraction, and each addressing mode corresponds to an applicative constructor for effective addresses. Each instruction, however, corresponds to an imperative procedure, which is executed to cause binary emission.
- There are other ways to mix alternatives. For example, one might construct abstract instructions, but emit their binary representations as side effects. Or one might construct binary representations (sequences of tokens) directly, allowing the application to concatenate them explicitly. The Toolkit supports the first mix but not the second.

Interfaces to decoders

For the top left mapping depicted in Figure 1, from binary code to procedures, the Toolkit generates a decoder that distinguishes all the instructions and addressing modes and calls the appropriate procedures in the client's interface. This kind of generated decoder is not very flexible; to use it, a client must specify an action for each instruction and addressing mode. This kind of decoder can also be inefficient; because it distinguishes all instructions and addressing modes, it may do more work than is needed to solve a particular problem. For example, to do control-flow analysis, it is not necessary to distinguish among different floating-point instructions, so an efficient decoder should not examine a floating-point opcode. As a better interface to its decoder-generation capability, the Toolkit provides a construct called a *matching statement*. A matching statement enables a programmer to build an efficient decoder that associates single actions with either individual instructions or groups of instructions. The term "matching statement" is motivated by our model of the decoding interface, which is "ML-like pattern matching over binary representations of machine instructions."

Figure 2 shows an example matching statement, which is embedded in a Modula-3 procedure. The text between `match` and `endmatch` is the matching statement. The phrases between vertical bars (`|`) and right arrows (`=>`) are *patterns* written in

[2] In a CISC instruction set, an effective address is a fragment of a machine instruction that identifies the location of an operand or a result. Typically this location refers either to a machine register or to a location in memory.

```
PROCEDURE Follow(m:Memory.T; pc:Word.T):FollowSet.T =
VAR succ : Word.T;
BEGIN
  match [succ] pc to
  | nonbranch                          => RETURN FollowSet.T{succ};
  | call(target)                       => RETURN FollowSet.T{target};
  | branch(target) & (ba | fba | cba)  => RETURN FollowSet.T{target};
  | branch(target)                     => RETURN FollowSet.T{succ, target};
  | jmpl(dispA(rs1, simm13), rd)       => RETURN FollowSet.T{GetReg(m, rs1)+simm13};
  | jmpl(indexA(rs1, rs2), rd)         => RETURN FollowSet.T{GetReg(m, rs1)+GetReg(m, rs2)};
  | some itoken                        => Error.Fail("unrecognized instruction");
  endmatch
END Follow;
```

Fig. 2. Matching statement used for control-flow analysis of SPARC instructions

the Toolkit's domain-specific specification language, SLED (Ramsey and Fernández 1997). These patterns describe the binary representations of either individual instructions or groups of instructions. The code on the right-hand sides of the arrows is ordinary Modula-3 code. When the matching statement is executed, the generated decoder finds the first pattern that matches the instruction stream at address `pc`, then executes the corresponding right-hand side—just as in ML pattern matching.

The code in Figure 2 is a simplified version of code used in a retargetable debugger to help implement breakpoints on the SPARC. The procedure returns the set of instructions that could be executed immediately after an instruction at which a breakpoint has been planted (Ramsey 1994). The version in the figure omits subtleties associated with delayed branches. Each case in the matching statement corresponds to a family of machine instructions. For example, the pattern `nonbranch` matches any non-branching instruction. In this case, the only instruction that can be executed immediately afterward is the inline successor, which is located at `succ`. The Modula-3 expression that is returned, `FollowSet.T{succ}`, represents the singleton set of addresses containing `succ`. In the next case, the pattern `call(target)` matches call instructions. This occurrence of `target` is a binding occurrence. Because control can flow only to the target address, again the Modula-3 code returns a singleton set. The third case matches unconditional branches, and the fourth case matches all branches not matched by previous cases, i.e., the conditional branches. Because a conditional branch can flow to its inline successor or to its target address, the Modula-3 code for the fourth case returns a set containing those two elements. The two `jmpl` patterns are indirect jumps through registers; the `GetReg` procedure gets the value in the register in order to compute the target address.

Printing assembly language

A procedure that maps to assembly language must emit strings, and the client must provide a suitable interface. To be suitable, an interface must conform to the idioms used for printing in the language of the generated code. This problem is more difficult than it might appear, because language paradigm alone does not determine the idiomatic way to print. Several examples illustrate this point.

The example printing procedure on page 8 shows the interface the Toolkit uses between generated printing procedures and clients written in C. The interface consists of two variables:

```
void (*asmprintf)(void *cl, const char *fmt, ...);
void *asmprintfd;  /* first arg to asmprintf */
```

Because C's rules for type casting are very loose, we can assign either of the standard library functions `fprintf` and `sprintf` to `asmprintf`. Of course, the C compiler does no useful type checking on these assignments. We made `asmprintf` and `asmprintfd` global variables not merely for convenience, but to enable the encoding procedures for both binary and assembly language to have identical prototypes.

The imperative idiom used to print assembly language in C has a counterpart in ML, but instead of using two variables, it is idiomatic to use a closure. For example, to print assembly language to standard output, one might define

```
val asmprintf = ref (fn s => TextIO.output(TextIO.stdOut, s))
```

Because ML is a functional language, however, many ML programmers would find it more natural to use an applicative interface that returns a string, or perhaps a list or tree of strings. To generate truly idiomatic ML, a program generator must provide a choice of printing interfaces.

Modula-3 is an object-oriented language, so you might expect that the idiomatic way to print would be to use an object type with a print method. The correct idiom is actually to use a fully abstract type: the "writer" type defined in the Modula-3 standard library. Although writers are implemented using object types, the methods of the object type are hidden from clients, which use only the procedures defined in the `Wr` interface. A suitable interface for emitting assembly language might say

```
VAR asmwr : Wr.T;
```

The details of the examples above are language-dependent, and they may seem obscure or picky, but they matter because *they are exposed at the interfaces* between generated code and client code. For example, a Modula-3 programmer who is forced to use a "printing object" instead of a writer is also forced to write glue code by hand in order to use the Modula-3 library. These kinds of requirements seem to call for very fine control over program generation. Other program generators must cope with similar problems. In the program generator for the Abstract Syntax Description Language, for example, one of the hardest problems has been to hide differences between target-language idioms, even for a problem as seemingly straightforward as printing.[3]

[3] Private communication from Dan Wang, 21 May 2000.

2.3 Controlling the output of the generator

The significant variations in the output of the Toolkit come from variations in the instruction set itself and in the interfaces to the encoders and decoders. Both the binary and the assembly-language representations of the instruction set are described using a domain-specific language: SLED, the Specification Language for Encoding and Decoding (Ramsey and Fernández 1997). A SLED specification captures knowledge in a specific domain (instruction formats), and writing one is a job for a domain expert. Because the SLED specification does not bear on the issues discussed in this paper, this paper does not include an example. Examples can be found in papers by Ramsey and Fernández (1995, 1997).

As noted above, a decoder's interface to an instruction stream is defined by an abstraction that has one type (address) and three operations: add an integer to an address, fetch a token at an address, and convert an address to an integer. To define an instance of this abstraction, a client of the toolkit does not use the mechanisms of the language in which the client is written (and in which the Toolkit generates code). Instead, the client uses a simple form of macros provided by the Toolkit. Client programmers provide "code templates" that perform the basic operations above, and the Toolkit instantiates these templates as needed.

As an example, here is the part of the SLED specification that gives the templates used in a binary translator, which is written in C (Cifuentes, van Emmerik, and Ramsey 1999); the instruction stream is implicit in the `getDword` operation.

```
address type       is    "unsigned"
address add        using "%a+%o"
fetch   32         using "getDword(%a)"
address to integer using "%a"
```

As another example, here are the templates used in a debugger, which is written in Modula-3 (Ramsey and Hanson 1992):

```
address type       is    "Word.T"
address add        using "Word.Plus(%a, %o)"
fetch   any        using "FetchAbs(m, %a, Type.I%w).n"
address to integer using "%a"
```

The variable `m` used in the fetch template stands for an instruction stream. A matching statement generated using this template will compile correctly only in contexts in which `m` stands for an instruction stream.

Variations in other interfaces are controlled by command-line options. These variations include applicative and imperative views of encoding, support for direct and indirect calls, and various conventions for naming procedures, types, and structures. To achieve even the incomplete control provided by the Toolkit, there are nine options that affect the interfaces to encoding procedures. Another three affect internals (e.g., logging), and perhaps a half dozen affect relocation procedures.

2.4 Assessment

Multiple languages

The only language for which the Toolkit provides full support is C. The Toolkit provides partial support for Modula-3, Standard ML, and Objective Caml; the λ-RTL Translator provides partial support for C, Standard ML, and Objective Caml. Section 4, which discusses the Toolkit's implementation, explains why the multiple-language support is limited.

Idiomatic and readable code

Code generated by the Toolkit or Translator respects common programming idioms in the target language. The most important idiom to get right is probably printing. The Toolkit also supports indirect calls through C structures, an idiom that is used to write object-oriented programs in C. These tools also provide a suitable choice of functional and imperative idioms for clients written in functional or imperative languages.

Code generated by the Toolkit or Translator is fairly easy to read. Most of the readability comes from using a code prettyprinter after the style of Oppen (1980). Some comes from other choices about the generated code; for example, the call to `emitm` on page 8 contains many constant expressions that could be simplified by the program generator. We elected not to simplify these expressions because the generated code is much more readable if they are left unsimplified.[4]

It is self-reinforcing for the author of a program generator to emit readable code; if the code is readable, it is easier to debug the program generator and to understand the performance and behavior of the generated code.

Control of interfaces and internals

Compared to a program generator such as Yacc or BURG, the Toolkit and Translator create code with many interfaces and even more variations on interfaces and internals.

Instruction streams To write an instruction stream, code generated by the Toolkit or Translator uses a single interface that is determined by the implementation language of the generated code. In our experience and that of our users, this single interface seems adequate to the task of emitting instructions quickly.

To read an instruction stream, code generated by the Toolkit or Translator uses code templates, as explained in Section 2.3. The code templates are a surprisingly effective mechanism: they leave the packaging of the instruction-stream abstraction open, and they support multiple languages effortlessly. Using templates, programmers can easily arrange for decoders to manipulate instruction streams without the

[4] By command-line option, a user can force the program generator to simplify all expressions.

overhead of procedure calls. Finally, there are no command-line options or configuration parameters to remember.

Users of functional languages may see a connection between templates and parameterized modules. For example, in Objective Caml one might try to parameterize a decoder by a module of the following type:

```
module type S = sig
  type address
  val add    : address -> int -> address
  val fetch  : address -> width:int -> int32
  val to_int : address -> int32
end
```

You might be tempted to conclude that the Toolkit's code templates are simply a poor man's substitute for parameterized modules, but a closer look shows two problems with encapsulating addresses in module parameters.

- The interface leaves implicit the instruction stream from which information is to be fetched. This information is intended to be hidden inside the implementation of `fetch`, but that means `fetch` should be a closure, not a top-level function. It's easy to make `fetch` a closure when using code templates (e.g., the template written in Modula-3 has the free variable `m`), but not when using module parameters.
- Functions such as `to_int`, `add`, and `fetch` are typically very small and should be inlined, but many implementations of functional languages don't inline functions that appear in module parameters. In low-level systems codes for which performance is critical, such as dynamic compilers and optimizers, the cost of a function call for such small operations would be unacceptable (Engler 1996; Bala, Duesterwald, and Banerjia 2000). Using templates, client code can achieve excellent performance even if using a compiler that does not inline procedure calls or method calls.

The code templates do not simply substitute for parameterized modules; they are actually better suited to the purpose.

Encoding procedures Interfaces to encoding procedures are controlled by a large number of command-line options. This situation is intolerable; even the Toolkit's authors are puzzled by the variations. As an alternative, I have considered a configuration language, but my enthusiasm is limited; it is not clear why should a big set of configuration variables should be significantly better than a big set of command-line options.

Could we solve the problem by eliminating variations? Our users say no. Applications that generate code dynamically, such as DyC (Auslander et al. 1996), Fabius (Lee and Leone 1996), VCODE (Engler 1996), and Dynamo (Bala, Duesterwald, and Banerjia 2000), all want the best possible performance. Overhead for indirect calls would be unacceptable. In the case of VCODE, even procedure-call overhead

```
module Machine {
  cell     = CELL  (space, exp)
  exp      = CONST (const)
           | FETCH (cell)
           | APPLY (operator, exp*)
  const = BOOL (bool)
        | INT (int)
  space = (char)

  char = (int)
  bool = (int)
  operator = (identifier) -- treat as opaque
}
```

Fig. 3. Example ASDL description

is unacceptable; VCODE cannot be implemented using the Toolkit because it requires macros.[5] Authors of other applications, like static compilers, prefer to call encoding procedures indirectly, so they can easily get both binary code (for speed) and assembly language (for debugging) from the same compiler.

Decoders Clients see no formal interface between an application and a generated decoder. Because a matching statement acts just like a statement in the source language (C or Modula-3), and because an action within a matching statement is an ordinary statement in the source language, the Toolkit effectively extends the language's abstract syntax with direct support for instruction decoding. Compared with code templates and command-line options, this syntactic extension is the simplest method of controlling how generated code is coupled to client code. Because the application programmer writes a matching statement wherever a decoder is needed, there is no need to wrap a generated decoder in a procedural abstraction, and therefore there is no need to provide different interfaces to different clients.

One penalty of the approach is that an error in a matching statement or in a template can be detected only *after* the program generator runs, and the resulting error messages may be mystifying. The Toolkit mitigates the problem by careful use of line-numbering pragmas (e.g., `#line` in C), which make it possible for some error messages to refer to source-code locations in the original matching statement.

3 Case study: The Abstract Syntax Description Language

The Abstract Syntax Description Language (ASDL) describes recursive, tree-like data structures, such as are used in compilers to represent abstract syntax, intermediate code, etc. (Wang et al. 1997). The corresponding program generator, `asdlGen`, is intended to solve three problems:

- To make it easy to manipulate trees using the implementation language of one's choice

[5] Private communication from Dawson Engler, 1995.

- To make it possible to write such trees to disk and read them back again
- To make it possible to exchange trees between programs written in different languages.

The core of ASDL is a data-definition language with a simple type system. ASDL's three base types are `int` (arbitrary-precision integer), `string` (sequence of characters), and `identifier` (atomic name with fast equality test). ASDL also has type constructors for products and for recursive sum types with named variants. Finally, ASDL has type constructors for lists and optional values. In a language like ML or Haskell, an optional value would be represented by a value of `option` or `Maybe` type; in a language like C or Java, an optional value would be represented by a pointer value that is possibly `NULL`.

Figure 3 shows an example ASDL description, which describes an oversimplified version of a data type used to represent low-level expressions in a compiler. The `cell` type represents a location (given by an expression `exp`) in a register or in memory (identified by a `space` of `'r'` or `'m'`). An expression is a constant, a fetch from a cell, or an application of an operator to a list of expressions. Constants may be Boolean or integral. Because ASDL does not support characters, Booleans, or opaque types, it is necessary to supply definitions for `char`, `bool` and `operator`.

3.1 What the generated code does

The code generated from an ASDL description can be divided into three parts, which describe trees, create trees, and serialize trees. (Serialization is the process of converting a data structure in memory into a serial representation that can be written to disk or sent over a network. It is also called flattening, pickling, and marshalling.)

Describing trees

To describe trees, ASDL defines a reasonable mapping from ASDL's type system to the type system of the target language. In Standard ML, for example, `asdlGen` relies on the language's direct support for sum types with named constructors. `AsdlGen` generates this representation of the `const` type from Figure 3:

```
signature Machine_SIG =
  sig
    datatype const =
          BOOL of (bool)
        | INT of (StdPrims.int)
    ...
    withtype bool = (StdPrims.int)
    ...
  end (* sig *)
```

In C, for example, the local idiom for sum types combines an untagged union with an explicit tag. The resulting type definition is more awkward than in ML:

```
typedef StdPrims_int_ty Machine_bool_ty;
typedef struct Machine_const__s* Machine_const__ty;
struct Machine_const__s {
    enum {Machine_BOOL_enum, Machine_INT_enum} kind; union  {
        struct Machine_BOOL_s {
            Machine_bool_ty bool1; } Machine_BOOL; struct Machine_INT_s {
            StdPrims_int_ty int1; } Machine_INT; } v; };
```

In Java, for example, the local idiom for sum types is to represent the sum type by a single "abstract base class," from which each alternative in the sum inherits. Here is a somewhat simplified version of the code:

```
public final class bool { public int int1;

    public bool(int int1) {
        this.int1 = int1;

    }
}
public abstract class const_ {
    ...
}
public final class BOOL extends const_ {
    public bool bool1;

    public BOOL(bool bool1)
    {
        this.bool1 = bool1;

    }
    ...
}
```

`AsdlGen` 2.0 supports many target languages: C, C++, Haskell, Icon, Java, Objective Caml, and Standard ML. For each target language except Icon, the ASDL manual explains how ASDL's type system maps to that language's type system.

Creating trees

To create trees, `asdlGen` uses one of two strategies. In a target language that has native support for creating tree nodes, such as Haskell or ML, `asdlGen` generates only the description of the tree type; to create a tree, a client program simply applies the constructors of the tree type. In a target language that has no native support for creating tree nodes, such as C, `asdlGen` generates procedures that allocate space for and initialize tree nodes; a client program calls these procedures to create trees applicatively. For example, here is a generated C procedure used to create a tree node that represents a Boolean constant of type `const`:

```
Machine_const__ty Machine_BOOL(Machine_bool_ty bool1)
  {
      Machine_const__ty t1_; t1_ = malloc(sizeof(*t1_));
      if(t1_ == NULL)
          die();
      t1_->kind = Machine_BOOL_enum;
      t1_->v.Machine_BOOL.bool1 = bool1;
      return t1_;
       }
```

Serializing trees

To serialize trees, ASDL creates two families of procedures: one to read and one to write. A serialized tree is called a pickle, a reading procedure is called an unpickler, and a writing procedure is called a pickler. Because the format of an ASDL pickle is independent of the programming language in which the trees are created, pickled, and unpickled, components implemented in a variety of languages can exchange pickles.

The generated code contains a pickler and an unpickler for each type in the ASDL description. Here, for example, are declarations of two unpicklers; the declarations come from a generated interface that is written in Standard ML:

```
val read_bool : StdPkl.instream -> Machine.bool
val read_const : StdPkl.instream -> Machine.const
```

Here is the implementation of `read_const`:

```
and read_const s_ =
    let
      val t = (StdPkl.read_tag s_)
    in (case (t) of
        1 => let
          val bool1 = (read_bool s_)
        in Machine.BOOL(bool1)
        end
      | 2 => let
          val int1 = (StdPrimsUtil.read_int s_)
        in Machine.INT(int1)
        end
      | _ => (StdPkl.die ()))
    (* end case *)
    end
```

Here are the declarations of the analogous procedures in the corresponding C interface:

```
StdPrims_int_ty Machine_read_bool(instream_ty s_);
Machine_const__ty Machine_read_const_(instream_ty s_);
```

3.2 Interfaces between generated code and client code

The examples above show not only what the generated code does but also what the interfaces look like. Two properties of these interfaces are noteworthy.

- Generated code can mix separate functions in a single interface. The original version of `asdlGen` always interleaved type definitions, constructors, and serialization code in a single interface. In some target languages, such as ML, `asdlGen` 2.0 places the serialization code in a different interface from the type definitions. In other target languages, such as C, `asdlGen` 2.0 places the serialization code in the same interface as the type definitions.
- In the code generated by `asdlGen`, the serialization code is coupled to the concrete representation of trees. It is not possible to generate code that is coupled to an abstraction representing trees. (As described below, it *is* possible to change the concrete representation that `asdlGen` uses.)

3.3 Controlling the output of the generator

For controlling the interfaces to and the internals of the generated code, ASDL provides two mechanisms: command-line options and views. Command-line options are self-explanatory; a *view* is a named collection of annotations, each of which attaches a property to a module, type, or constructor defined in an ASDL specification.

A command-line option passed to `asdlgen` can control one of a few properties of generated code:

- The language in which generated code should be written
- The number of columns into which the text of generated code should fit
- The location in which generated code should find supporting library code
- The representation of lists that should be used in generated code
- The representation of integers that should be used in generated code
- The view that should be used to control other aspects of generated code

An ASDL view can attach any number of *properties* to any named module, type, or constructor in an ASDL description. A property is simply a key-value pair; to have any useful effect, the key must be known to `asdlGen`. The uses of views are manifold and difficult to characterize precisely. The most significant uses are to override default choices of names and representations and to insert hand-written code in various places in the generated code. This paper presents some examples, but the whole story is told only in the manual (Wang 1999).

An example of a property that can be changed with a view is the `natural_type` property, which determines the generated code's representation of an ASDL type. If a view changes this property, it also must provide a bijective mapping between the default representation and the new representation specified by the view. This mapping is specified by a pair of properties: `wrapper` and `unwrapper`. Figure 4 shows a view that changes the `natural_type` property. The view changes the representations of the `char` and `bool` types that are defined in Figure 3. The mapping between characters and integers can be taken directly from the `Char` module in

```
view SML {
 module Machine <= implementation_prologue
%%
fun int2bool i = i <> 0
fun bool2int p = if p then 1 else 0
%%
  Machine.char <= { natural_type : Char.char
                 wrapper : Char.ord
                 unwrapper : Char.chr
               }
  Machine.bool <= { natural_type : Bool.bool
                 wrapper  : bool2int
                 unwrapper : int2bool
               }
}
```

Fig. 4. A view that changes representations of `char` and `bool`

the initial basis of Standard ML, but the mapping between Booleans and integers requires the hand-written functions `int2bool` and `bool2int`. Figure 4 uses the `implementation_prologue` property to insert definitions of these functions into the generated code.

When this view is used with the ASDL specification in Figure 3, the description of the tree type `const` is changed as follows:

```
signature Machine_SIG =
  sig
    datatype const =
          BOOL of (Bool.bool)
        | INT of (StdPrims.int)
    ...
    withtype bool = (StdPrims.int)
    ...
  end (* sig *)
```

Although the *use* of the ASDL type `bool` within the `const` type uses the new definition, the signature retains the old *definition* of `bool` as `StdPrims.int`. The view also changes the types of the generated functions `read_bool` and `write_bool`, and `asdlGen` inserts `int2bool` or `bool2int` in each function as appropriate:

```
    and read_bool s_ =
        let
          val int1 = (StdPrimsUtil.read_int s_)
        in (bool2int (int1))
        end
```

Views can achieve a variety of other ends; here is a partial list.

- A view can suppress program generation for one module, so that a user can supply a hand-written implementation of that module.

- A view can change the name that generated code uses for a module or a constructor.
- A view can change the name of the base class that is generated to represent a sum type in an object-oriented language.
- A view can replace the body of a serialization function (reader or writer) with a call to another named function.
- A view can specify the value of the integer tag associated with a sum-type constructor in generated C, C++, or Java code.

3.4 Assessment

Multiple languages

`AsdlGen` provides impressive support for multiple languages. If a user can live with the default representations, he or she can achieve instant data exchange between programs written in any of seven languages. Although ASDL has aspects of a lowest-common-denominator system, it does provide type constructors, notably lists, that are not found in all target languages.

Idiomatic and readable code

ASDL is designed to support one programming idiom: applicative construction and deconstruction of trees. This idiom maps very naturally onto such languages as ML and Haskell, and the ASDL documentation does a good job explaining how the idiom works in other target languages. In the details, however, `asdlGen` generates code that is unidiomatic or unreadable in its use of names, concrete syntax, interfaces, and types. Even more significant are missed opportunities to support separate compilation and abstraction.

Names It can be difficult for a program generator to avoid name collisions when emitting code in languages such as C, where there is a single, flat name space at top level. An easy solution is to extend the generated names with enough prefixes and suffixes to make the names unique. Sometimes prefixes or suffixes can be idiomatic; Hanson (1996), for example, uses a C programming idiom in which a "module" name is used as a prefix. Unfortunately, `asdlGen` overuses prefixes and suffixes, which makes generated code hard to read. For example, the `_ty` suffix on type names is not idiomatic C, and this suffix is not needed to avoid collisions. As another example, the `Machine_` prefix on the name of the union member `Machine_BOOL` is not necessary or idiomatic. Because the members of a union occupy their own private name space, it is safe to use `BOOL`, the name of the datatype constructor.

There are other, minor examples of poor use of names in code generated by `asdlGen`. A local variable in C code or ML code may have an unnecessary underscore at the end of its name. The names of the serialization procedures in ML use internal underscores (e.g., `read_bool`) where the Standard Basis capitalization conventions recommend mixed case (e.g., `readBool`).

Concrete syntax As can be seen in the examples above, concrete syntax generated by `asdlGen` can suffer from two flaws: excessive use of parentheses and poor choice of indentation and line breaks. These problems might seem trivial, but programmers tend to avoid a program generator if they can't easily read its output—especially the interfaces. It is surprisingly difficult to do a really good job putting indentation and line breaks in generated code, and even avoiding redundant parentheses is not completely trivial (Ramsey 1998).

Interfaces The interfaces generated by `asdlGen` typically mix type definitions, creation functions, and serialization functions. When a type definition exposes its representation, it is more idiomatic and readable to put related utility functions in separate interfaces. While any module using the generated code will need access to the type definition, most will use either creation functions, reading functions, or writing functions—not all three. Some clients may use *only* the type definition and no generated operations. Merging all into one interface makes it harder to understand and use the generated code, even if one never looks at the implementation.[6] Worse, it creates a configuration problem: code generated with `asdlGen` can't be compiled without an installation of `asdlGen` that provides serialization libraries, and installing `asdlGen` can be burdensome because the installation procedure tries to build libraries for all the languages that ASDL supports.

Types ASDL's use of types should be assessed at two levels: the level of the ASDL type system and the level of the generated code. ASDL's type system includes many type constructors of proven value. Sums, products, lists, and optional types suffice to support a rich variety of common programming idioms. Since ASDL is intended to describe data, not code, it is sensible that ASDL does not support function types or object types. The choice of base types for ASDL is less understandable. While strings, symbols, and unlimited-precision integers are very useful, there seems to be no good reason for omitting characters and Booleans. The most significant omission, however, is ASDL's lack of support for abstract types; data abstraction is an essential programming idiom, and almost all of `asdlGen`'s target languages support it in some form.

It is possible to simulate an abstract type using an ASDL view. In the main part of the ASDL description, the programmer must pretend that the abstract type is some concrete type, e.g., a string. The programmer can use a view to override the pretend type with the actual type and to provide conversion routines between the pretend type and the actual type. These conversion routines must be written even

[6] Readers unfamiliar with compiler construction may benefit from some elaboration of these claims. Today's compilers are often written using many simple passes. Many such passes might analyze or transform a data structure defined by ASDL, such as an abstract-syntax tree or an intermediate-representation tree, but very few passes need to worry about serialization. The author of a register allocator, for example, will at best be distracted if the interface refers to functions and types that are used only for serialization. The problem can be mitigated if the program generator carefully separates different parts of the interface, for example by putting common type definitions first and serialization code last. In effect, one then has several interfaces that just happen to be written in one file—a tenable situation.

if the programmer knows that values of the abstract type will never be serialized, because there is no way to assert that such values will never be serialized.

At the lower level, `asdlgen` does a very good job mapping ASDL's types to types in the generated code, and this mapping is carefully explained in the manual. This achievement is especially praiseworthy because the ASDL type system does not take the easy way out of using only type constructors that are available in all target languages. The ASDL system fills the gaps with libraries; for example, `asdlGen` comes with list libraries for C, C++, and Java.

There is one minor bug in `asdlGen`'s low-level treatment of types; as shown on page 22, changing the `natural_type` property changes all the uses of an ASDL type, but it leaves a misleading definition in the generated code.

Separate compilation `AsdlGen` does not support separate compilation; all types used in a representation must be presented to a single run of `asdlGen`. This restriction inhibits a number of useful programming idioms. For example, a programmer may want to define a type using hand-written code, e.g., to exploit a feature that is available in the target implementation language but not in ASDL. Code generated by `asdlGen` cannot easily refer to such a type, even when there is hand-written serialization code available to support it.

A related problem, which is one of the most significant defects in ASDL, is that it is difficult to define a tree type that uses a type defined in another source file. For example, if a programmer is designing a compiler, he might want to define the statement type and the expression type in different source files. Because a statement may contain an expression, this structure is difficult to achieve using ASDL, even if both types are defined in ASDL.

The program generator would benefit from a cleaner way to interface to separately compiled types, for which the definitions, constructors, picklers, and unpicklers are in ordinary source files. Such source files might be generated by other runs of the generator, or they might be written by hand.

Abstraction One way to think of an ASDL pickle is that it says which constructors should be called, in what order, to reproduce a tree. If we think abstractly, a pickle is a specification for a catamorphism: a higher-order function that builds trees. Just as *foldr* implements abstract list construction using any suitably typed functions in place of *nil* and *cons*, so should `asdlGen` generate an unpickler that implements abstract tree construction using any suitably typed functions in place of the concrete constructors that `asdlGen` defines for the tree. This conclusion is also supported by the algebraic approach to data abstraction; because there is an obvious one-to-one mapping between trees of some concrete algebraic data type and a corresponding abstraction, the concrete representation should be irrelevant (Liskov and Guttag 1986).

As an example, here is an interface to such an unpickler for trees of type `const`:

```
val read_const_catamorphically :
  { bool : StdPrims.int -> 'a, int : StdPrims.int -> 'a } ->
  StdPkl.instream -> 'a
```

Such an interface could easily be produced mechanically; we require a parameter of function type for each datatype constructor, and we replace all occurrences of the `const` type with the type variable `'a`. In the implementation of `read_const` on page 20, we simply replace `Machine.BOOL` and `Machine.INT` with the function parameters.

Unfortunately, the designers of `asdlGen` overlooked the possibility of catamorphic interfaces. An unpickler generated by `asdlGen` cannot call abstract tree-creation functions; it always creates the concrete representation that corresponds to the ASDL type. The only way to pickle and unpickle a value of an abstract type is to write "transducers" by hand. The `int2bool` and `bool2int` functions in Figure 4 are examples of such transducers; although they work with the concrete type `Bool.bool`, they could as easily work with an abstract type. One can interleave the execution of a transducer with the execution of a generated pickler or unpickler by suitable use of the `wrapper` property in an ASDL view, but the wrappers do not eliminate the need for transducers. The interleaving does not eliminate the use of ASDL's concrete representation as a bridge between the serial representation and the client's abstract representation, but it does reduce the lifetime of the concrete representation. An implementation based on catamorphisms would automatically be deforested: it would never create values of the concrete representation.

Unhappily, transducers are needed not only for abstraction but also to interface to existing compilers. Hanson (1999) reports on experience with ASDL and `lcc`.

> This duplication of effort [the transducer] is an onerous side effect of retrofitting an existing compiler with ASDL... Most of the code in the ASDL back end is devoted to building copies of [the compiler's] data structures—that is, building a different, but logically equivalent representation for nearly everything.

Christian Lindig, Simon Peyton Jones, and I have found that transducers are necessary even when designing a new compiler from scratch. We want to couple a back end to front ends written in different languages, so we want the ability to pickle and unpickle. Inside the compiler, however, we want not to use the representation defined by ASDL; instead we want a representation that includes private annotations: sets of registers read and written, sets of live variables, and so on. Such annotations, which are likely to change as the compiler evolves, should not be visible from the external interface used by the pickler, which should not change as the compiler evolves. Because we cannot couple `asdlGen`'s generated code to an abstraction, we must write transducers.

These needs for transducers arise not from fundamental limitations of program generation or of ASDL, but from an accident of `asdlGen`'s design. `AsdlGen` could be modified to emit a type-definition interface giving only an abstraction, not a representation. It could also emit picklers and unpicklers parameterized by the abstraction; `read_const_catamorphically` is one example of a parameterized unpickler.[7] The flaw in the program generator lies not in the fundamental model or the underlying algorithms, but in the generator's lack of support for an important programming idiom.

Control of interfaces and internals

Three aspects of the control of the ASDL program generator are worth assessing: the mechanisms used for control, the choices provided for interfaces, and the choices provided for representations.

Mechanisms `AsdlGen` uses command-line options and views to control interfaces to and internals of generated code. These mechanisms are most comparable to the command-line options that the Toolkit uses to control generated encoding procedures, and they suffer from one of the same problems: complexity that users find daunting and difficult to manage. The view mechanism, however, does have two properties that make it superior to command-line options alone.

- The view is part of the same source language as the ASDL description, and it is processed by the same processor. Unlike a collection of command-line options stored in a Makefile, a useful view is much less likely to be lost or overlooked.
- Using a view, a programmer can collect a set of useful directives and give the set a name. He or she can then select this set by using the `-V` option on the command line. To be able to define such a shorthand is convenient. It is unfortunate that neither the view language nor `asdlGen`'s command line makes it possible to compose views.

Choices of interfaces `AsdlGen`'s mechanisms for controlling interfaces and internals are similar to the Toolkit's, if slightly superior. The range of possible interfaces, however, is more limited. As discussed above, the current range of interfaces precludes some important programming idioms: generated code cannot interface cleanly with abstract types, and serialization works only with `asdlGen`'s concrete representation. For serialization, at least, it is clear that some choice at program-generation time is needed. Many clients will be content to manipulate a concrete representation, but some clients, to avoid transducers, will want an interface based on catamorphisms. For the other problems noted above—abstract types and separate compilation—it is not clear that choices need to be provided at program-generation time; there may be a general solution that can be "always on."

[7] Granted, it is not obvious what mechanism to use for the parameterization, especially in C. Indeed, even though many object-oriented and functional languages provide natural, idiomatic parameterization mechanisms (e.g., classes, functors, generics, and templates), it is not clear that these are always the best mechanisms for a program generator to use. Still, any parameterization mechanism is better than none.

Internal representations `AsdlGen` does offer important choices in representation, especially representations of lists. These choices are worth discussing because questions of list representation arise not only in `asdlGen` and in the Toolkit but also in parser generators and other program generators. In all cases, the problem to be solved is to make generated code interoperate with hand-written code.

In some target languages, like ML and Haskell, polymorphic list types provide natural representations of lists. In other languages, there is no obvious natural representation. In C, for example, there are at least three alternatives: linked list, null-terminated array of pointers, and dynamic-array abstraction. Different applications have different needs; for example, the `lcc` back end needs an implementation in which appending to a list takes constant time (Hanson 1999).

When generating code in a monomorphic language, a user of sequences must decide not only what is a suitable representation but also how many types to define. *Heterogenous* generation defines a specialized sequence type for each different kind of sequence. This choice bloats the code, but it preserves type safety and keeps dynamic instruction counts down. *Homogeneous* generation uses a generic sequence type that contains something like C's `void *` or Java's `Object`. This choice does not duplicate code, but depending on the language it may abandon type safety (C) or require dynamic type checks (Java). Again, no single choice is always suitable; for example, when translating polymorphic types into Java, the Pizza translator provides both alternatives (Odersky and Wadler 1997).

Users of `asdlGen` make these kinds of choices with command-line options. The choices available depend on the language in which code is generated. The documentation is not completely clear, but it appears a user generating C code can choose between heterogenous and homogeneous generation, and a user generating Java code can choose between a linked-list representation and the built-in `java.util.Vector`. It is unclear why `asdlGen` provides only this limited set of choices; users might prefer to choose both representation and generation strategy independently, for all relevant languages.

Choosing representations of even simple, monomorphic types can be hard for the author of a program generator. Should integers have arbitrary precision (bignums), some fixed precision (e.g., 32 bits), or the native precision of integers in the target language (not specified in C, and perhaps 31 or 63 bits in a functional or object-oriented language that uses tags for garbage collection)? Should strings be null-terminated, or should they carry an explicit length? What codes should be used to represent the enumerations that `asdlGen` uses to distinguish different constructors? How can one match codes used in an existing application? How can one make codes distinct across *all* types, so each code identifies not only a constructor but also a type? Again, `asdlGen` solves some of these problems. By selecting the appropriate library, a user can choose between arbitrary-precision and limited-precision integers. By using the `enum_value` property in a view, a user can assign codes explicitly to constructors. A uniform, comprehensive mechanism would be preferable.

To summarize, different clients need different representations of ASDL's types, and there are surprisingly many choices. Finding a simple, clear way even to *specify* the clients' needs, let alone satisfy them, appears to be an unsolved problem.

External representations `AsdlGen` provides no choices about the representation of pickles (the serial format); the generated code uses a single representation. This limitation is sensible: the pickle format is a private matter that only generated code depends on, and using a single, fixed format helps guarantee compatibility between programs written in different languages. The pickle format has been designed with careful attention to performance; for example, a small integer is represented using only eight bits. At need, a user can effectively extend the format by using a view to replace picklers and unpicklers. No other choices are needed at program-generation time.

4 Implementations

It is not yet known how to build a program generator that promotes reuse by making it easy to provide multiple interfaces to generated code, in multiple languages. To solve this problem requires not only suitable ideas and abstractions but also a suitable internal structure for the program generator. The design of a program generator's intermediate representation influences its ability to support multiple languages and multiple interfaces. This section discusses, at a high level, the intermediate representations used in two implementations of the Toolkit and one implementation of `asdlGen`. The purpose of the discussion is to identify parts of these program generators that are not domain-specific and therefore might contribute toward a generally useful infrastructure for building reusable program generators.

4.1 Implementations of the Toolkit

The Toolkit has two implementations. The original, official implementation is written in Icon (Griswold and Griswold 1996). Mary Fernández and I chose Icon because it is an excellent match for the most difficult domain-specific problem of our original decoder generator: writing a backtracking, heuristic search for a good decoder. As we added support for multiple languages and for alternative interfaces, and as the program generators grew more complex, Icon's lack of static typing and lack of modules made the implementation more and more unwieldy until finally it became intractable. Icon was a poor match for handling the pragmatic aspects of program generation, the importance of which we had underestimated.

I undertook a new implementation of the Toolkit in Standard ML, where I could exploit static typing, polymorphism, and parameterized modules. I had two goals: to write new, clean implementations of the algorithms, so we could add improvements, and to build a serious infrastructure to support the pragmatic aspects of program generation. I have used the infrastructure not only for the Toolkit, but also for the λ-RTL project (Ramsey and Davidson 1998).

4.2 The Icon toolkit: trees and templates

Figure 5 shows the structure of the back end of the Icon implementation of the Toolkit. Words written in boxes depict intermediate forms, i.e., data representations

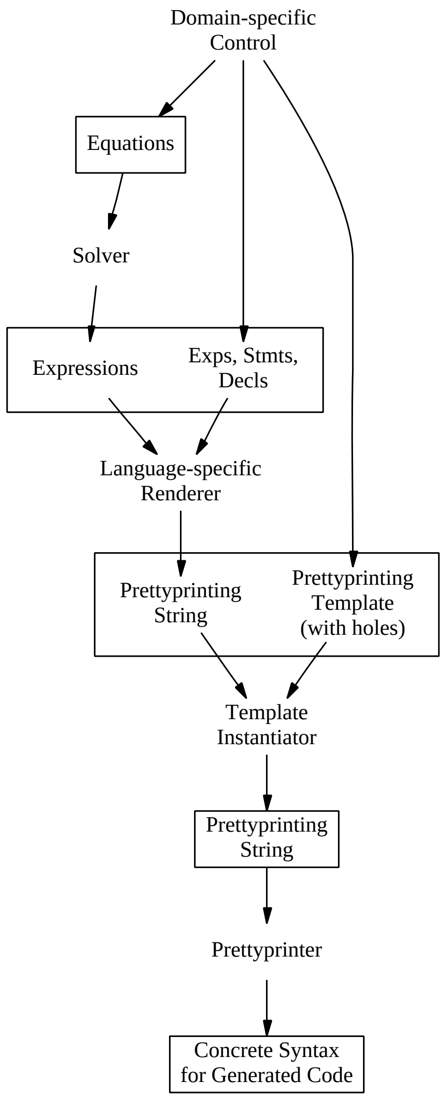

Fig. 5. Back end of the Toolkit (Icon implementation). Intermediate forms are boxed; components are unboxed.

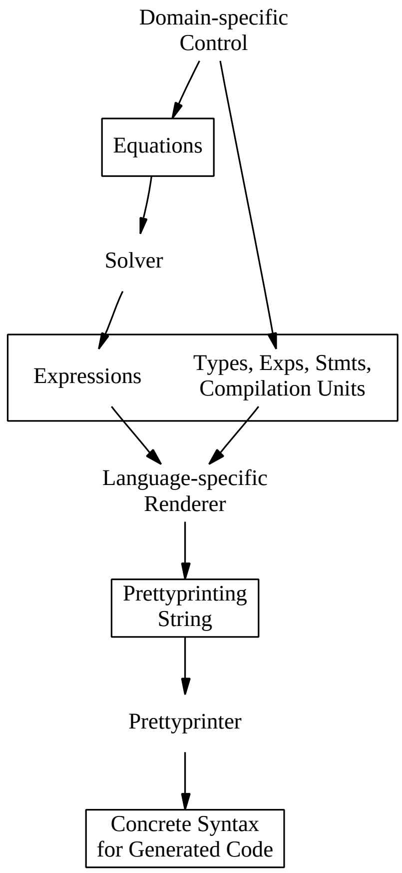

Fig. 6. Back end of the Toolkit (ML implementation). Intermediate forms are boxed; components are unboxed.

manipulated by the back end. Words not in boxes depict components of the back end. By studying relationships between intermediate forms and components, we can identify forms and components that might be reusable.

The front end of the Toolkit reads a machine description and expresses its domain-specific knowledge in the form of equations that relate abstract machine instruc-

```
procedure emit_create_instance_body(pp, cons)
  a := []
  every i := inputs_of(cons).name do
     put(a, template_to_list("instance-assignment.t", "l", i, "r", i,
                             "name", Cnoreserve(cons.name)))
  emit_template(pp, "create-instance-body.t",
                    "safename", Cnoreserve(cons.name),
                    "name", cons.name, "type", cons.type.name,
                    "args", arg_decls(cons), "assignments", a,
                    "class", if \indirectname then "static " else "",
                    "input-tests", input_width_tests(cons))
  return
end
```

Fig. 7. Code that instantiates an encoding-procedure template

tions and binary representations. The encoder generator uses an equation solver to compute the binary representation as a function of an abstraction. The decoder generator uses the same equation solver to compute the values of pattern variables (e.g., arguments to abstract constructors) in terms of the binary. The solver is intended to be reusable apart from the rest of the Toolkit (Ramsey 1996b).

The answers from the equation solver are bindings of expressions to names. Most expressions are represented as abstract-syntax trees, but to simplify the solver, linear combinations are represented as tables in which the keys are expressions and the values are coefficients. This representation is used throughout the Toolkit's back end, which was a mistake; it would have been better to hide the table representation inside the solver and to use only trees everywhere else. As it is, the idiosyncratic representation of linear combinations makes it unnecessarily hard to reuse the representation.

Expressions, statements, and declarations comprise the highest-level intermediate form in the Toolkit. This form is independent of the target programming language. Expressions, statements, and declarations are general-purpose ideas, but identifying a good set of constructs for general-purpose use is a significant unsolved problem, as is how best to extend such a set for use in any particular program generator. Even within the Toolkit, the encoder generator and decoder generator have significantly different needs, as some examples illustrate.

Like expressions, statements are represented as abstract-syntax trees. The encoder generator uses relatively few kinds of statements: sequence, conditional (for conditional assembly), case statement (for analyzing abstract effective addresses), and token emission. It also uses statements that announce errors and emit relocation information. The decoder generator uses more kinds of expressions, statements, and miscellaneous constructs. Additions include range tests, blocks with local variables, assignments, declarations of initialized data (arrays of names of instructions matched), comments, and line-number directives (`#line`). Many of these constructs would be useful in a general-purpose infrastructure for program generation; some would not. Thus, although the Toolkit's expression, statement, and declaration trees could be reused in another program generator, they include Toolkit-specific constructs that would be useless in another setting, and we must assume that they lack domain-specific constructs that would be needed in another setting.

The Icon implementation of the Toolkit does not use an abstract-syntax tree to represent a procedure definition or a compilation unit. Instead, it uses a "prettyprinting template," which is a combination of a prettyprinting string and a template. A prettyprinting string is a string that contains special markup that controls insertion of indentation and line breaks. A template is a string that contains "holes" that can be filled with other strings. A prettyprinting template, then, is a prettyprinting string that contains "holes" that can be filled with other prettyprinting strings.

As an example, here is the prettyprinting template for an applicative encoding procedure, which returns a value of an abstract type representing an "instance" of an instruction. The name of this template is `create-instance-body.t`.

```
%{class}%{type}_Instance %safename(%args) {$t
%{type}_Instance _i = { %{name}_TAG };
%{input-tests}%{assignments}return _i;$b
}
```

This template at once selects a style of interface, a target language, and concrete syntax. The `%` signs mark holes that are to be instantiated; the marks `$t` (tab) and `$b` (backtab) are escape sequences that control prettyprinting.

Figure 7 shows the code that uses the template to emit an encoding procedure. The parameter `pp` is a prettyprinting stream, which is like an output stream, except that it recognizes escape sequences and converts them to suitable indentation and line breaks. The `cons` parameter contains information about the constructor being emitted. Each assignment to an instance variable is created by instantiating the template named `instance-assignment.t`, which contains the string "`_i.u.%name.%l = %r;`." The functions `template_to_list` and `emit_template` instantiate templates; they are generated automatically by a single-use program generator. They are represented in Figure 5 by the words "Template Instantiator." The function `Cnoreserve` mangles the name of the instruction to ensure that it doesn't collide with any of C's reserved words. The flag `indirectname` controls the visibility of the generated procedure. If the encoding procedure is intended to be called indirectly, through a function pointer, it is given storage class `static`, so it won't be visible outside the generated compilation unit. Otherwise, it is given the empty storage class and so is visible to other units.

Because `emit_template` is passed a prettyprinting stream, when it instantiates the template it passes the resulting prettyprinting string on to the prettyprinter. As suggested in Figure 5, the output from the prettyprinter is nicely indented concrete syntax.

To embed an expression or statement into a procedure, the Toolkit renders the expression's or statement's abstract-syntax tree into a prettyprinting string, then uses the prettyprinting string to fill a hole in a template. The rendering necessarily depends on the target programming language, because the prettyprinting string contains concrete syntax in that language. The Icon implementation of the Toolkit includes two language-specific renderers: a C renderer that is 330 lines of Icon and a Modula-3 renderer that is 243 lines.

The template instantiator and the prettyprinter are reusable components. They have simple internal interfaces, and I have reused them in several program generators. The language-specific renderers are also reusable components, at least in principle, but their utility is limited because the input language they accept—the abstract syntax of expressions, statements, and declarations—is not really reusable.

Despite the presence of several reusable components, the structure of the Icon implementation, as shown in Figure 5, is not very good from the point of view of the requirements in Section 1.2.

- Knowledge of the target programming language is too widely distributed. To support a new programming language, we must write not only a new renderer but also new prettyprinting templates.
- Because templates are source code, each template determines the language of, the interface to, and the internal structure of the generated code. If any *one* of these things varies, we have to write new templates. The encoder generator, which supports only C, uses about 20 templates, which are 1–5 lines of code each.[8] As users demand more and more variations in the generated code, templates are too difficult to maintain.

4.3 The ML Toolkit: a more reusable design

Rewriting the Toolkit in ML provided an opportunity to design more reusable intermediate forms. In the new implementation, there are no language-dependent templates; the domain-specific control works only with language-independent abstract-syntax trees, as shown in Figure 6. In addition to expressions and statements, the ML implementation uses an explicit representation of the types to be used in the generated code. To replace templates, it uses *compilation units*, which generalize the declarations used in the Icon implementation. All four kinds of abstract-syntax tree can be divided into two classes: one for general purposes and one specific to the Toolkit.

- The general-purpose types are integers (signed and unsigned, and of arbitrary widths), Booleans, strings, characters, records, arrays, both safe and unsafe unions, functions, pointers, named abstract types, a unit type (equivalent to C's `void`), and objects with inheritance. This large type system is intended to cover many target languages.
  The Toolkit-specific types are effective addresses and instances of machine instructions. There is also a Toolkit-specific type constructor, which can make integer types "relocatable"; values of relocatable types have later binding times than values of ordinary integer types.

[8] The decoder generator, which supports both Modula-3 and C, uses only abstract-syntax trees, not templates.

- The general-purpose expressions include introduction and elimination operations for the general-purpose types, as well as typical integer and Boolean operations. There is also an expression construct that suppresses algebraic simplification; this construct is used to make generated code more readable.
  The Toolkit-specific expressions include introduction and elimination operations for the Toolkit-specific types. They also include a variety of operations for narrowing and widening (sign-extending) integer values and for testing whether a value n can fit in k bits.
  Unlike the Icon representation, the ML representation of expressions is not warped by the equation solver. Although the equation solver requires ordered linear combinations (Derman and Van Wyk 1984), they are used only within the solver, which uses about 30 lines of code to convert back and forth.
- The general-purpose statements include assignments, conditionals, case statements, block comments and commented statements, nested blocks with local variables, return statements, and an empty statement.
  The Toolkit-specific statements include a statement that emits a token, one that discriminates on an abstract instruction, one that branches to an arm of a matching statement, and so on.
- Compilation units include general-purpose constructs, which can declare and define procedures, types, variables, constants, and exceptions. Declarations are collected into *interfaces*; definitions are collected into *implementations*, which import and export interfaces. Both interfaces and implementations can be parameterized.
  The interface abstraction has a few Toolkit-specific values that represent hand-written code, including an encoding library, standard I/O, support for emitting assembly language, a sign-extension routine, and other miscellany.

The representations of types, expressions, statements, and compilation units comprise the intermediate form of the Toolkit. As does the original, this intermediate form mixes general-purpose constructs with Toolkit-specific constructs, making the infrastructure more difficult to understand and reuse.

Figure 6 shows that eliminating templates simplifies the back end. The domain-specific control manipulates only abstract-syntax trees, from which a language-specific renderer produces prettyprinting strings, from which a prettyprinter produces concrete syntax.

Because the new intermediate form can express many kinds of compilation units, it is more reusable than code templates. The increased generality of the new form has a cost, however. The language-specific renderers for C and for ML are about 600 and 800 lines respectively, or about double the size of the renderers in the Icon implementation. These numbers are probably affected most by the change of implementation language, by the larger domain of the new emitters (not just expressions and statements but also types, interfaces, and implementations), and by the elimination of templates, which can emit prettyprinted code very concisely.

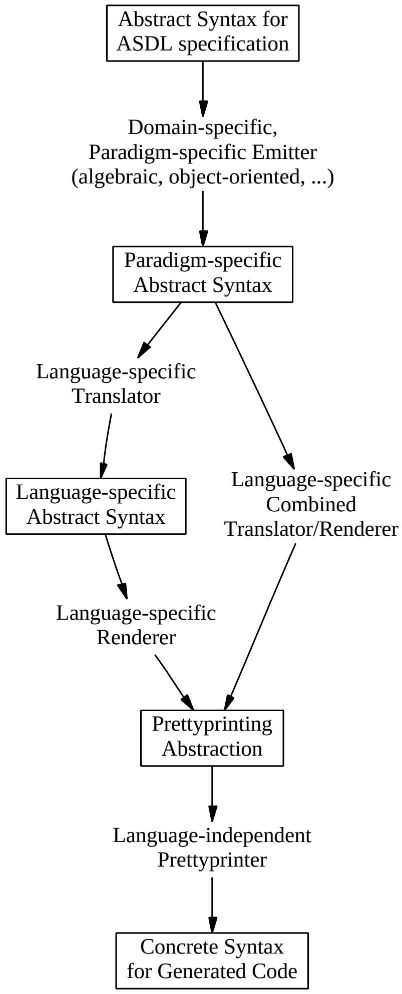

Fig. 8. Back end of `asdlGen`. Intermediate forms are boxed; components are unboxed.

4.4 Intermediate forms in `asdlGen`

`AsdlGen` uses a different design to support multiple languages. Compared to the Toolkit, `asdlGen` is clearly better at supporting multiple languages; it supports seven, not two or three. I cannot make detailed comparisons that show exactly how `asdlGen` achieves its superiority: although my knowledge of the Toolkit is intimate, my knowledge of `asdlGen` is limited by what I can learn from the code. Even a relatively high-level view of `asdlGen`, however, can suggest how to use intermediate forms to make a program generator reusable.

Figure 8 shows the intermediate forms used in the back end of `asdlGen`; it should be compared with Figures 5 and 6. Compared to the Toolkit, `asdlGen` achieves its multiple-language support by using more layers of intermediate form and by dividing its functionality into more components, which do simpler jobs. In the most general case, which is shown by the left path through Figure 8, `asdlGen` instantiates itself for a target language by composing three components: a paradigm-specific emitter, a language-specific translator, and a language-specific renderer.

- A paradigm-specific emitter makes high-level decisions about how to map ASDL's type system onto types in the target language. Supported paradigms include an algebraic paradigm, of which languages Haskell, Objective Caml, and Standard ML are instances, an Algol-like paradigm, of which C is an instance, a dynamically typed paradigm, of which Icon is an instance, an object-oriented paradigm, of which C++ and Java are instances, and a miscellaneous paradigm, which appears to provide some support for expressing ASDL pickles as HTML or XML documents. An example of a decision based on paradigm is that the algebraic paradigm maps an ASDL sum type to an algebraic datatype, the object-oriented paradigm maps a sum type to a class with subclasses, and the Algol-like paradigm maps a sum type to a tagged union.
  Each paradigm comes with an emitter and an abstract-syntax representation which are specialized to the paradigm but are independent of the particular language used within the paradigm. Emitters range in size from 170 to 353 lines of Standard ML.
- A language-specific translator maps a paradigm's abstract syntax onto language-specific abstract syntax. There is at most one translator for each language. As an example, the translator for C is 263 lines of Standard ML.
- A language-specific renderer converts a language's concrete syntax to a prettyprinting abstraction. This abstraction, which represents prettyprinted concrete syntax, is analogous to the Toolkit's prettyprinting string; it is based on work by Wadler (1999). There is at most one renderer for each language. As an example, the renderer for C is 389 lines of Standard ML.
- For some languages, the translator and renderer are combined into a single component, as shown by the right-hand path through Figure 8. As examples, each of the algebraic languages Haskell, Objective Caml, and Standard ML uses such a combined component. These components range in size from 206 to 254 lines of Standard ML.

To add support for a new language, one must design an abstract syntax and write a renderer for it. One must also select an appropriate paradigm and write a translator from that paradigm into the abstract syntax of the new language. Alternatively, the translator and renderer can be combined into one component, in which case no new abstract syntax is needed. If no existing paradigm is suitable for the translation, a new one must be designed, and an emitter for the new paradigm must be written.

The appropriate place to compare the Toolkit and `asdlGen` is at the mapping from language-dependent information to a prettyprinting abstraction representing concrete syntax. The Toolkit performs this mapping in a single step: the step labelled "language-specific renderer" in Figures 5 and 6. `AsdlGen` performs the analogous mapping in three steps: the emitter, translator, and renderer shown in Figure 8. `AsdlGen` can perform the mapping in two steps if the renderer and translator are combined. When renderer and translator are not combined, the language-specific abstract syntax and language-specific renderer are very likely to be reusable, because they are mostly independent of the specifics of `asdlGen`. Even when renderer and translator are combined, `asdlGen`'s design makes the implementor's job easier, because the emitter can operate without concern for low-level details of the concrete or abstract syntax of the target language.

5 Conclusions, speculation, and lessons learned

Authors of program generators should explore not only the design space of possible generated code but also the design space of possible interfaces to that code. In other words, authors should consider *interfaces*, not just implementations, when modeling the domain for which code is to be generated. Were I to attempt another reusable program generator, I would undertake discount usability tests (Nielsen 1993) with potential users. ("Please write on the board the code you would like to use to exploit my new frobozz generator.") Until then, it is possible to speculate about better ways of specifying interfaces and to draw a few conclusions about implementations.

5.1 Lessons for controlling interfaces and internals

The Toolkit uses three mechanisms to control interfaces to generated code. To specify the interface to a decoder, the programmer puts a matching statement at an appropriate place in the program; the syntax of the target language is effectively extended with matching statements. To specify the interface to an instruction stream, the programmer provides code templates, which are instantiated by the program generator. To specify the interface to an encoding procedure, the programmer chooses a set of command-line options. The first two mechanisms are easy and pleasant to use. Because there are too many variations (applicative vs. imperative, direct vs. indirect, objects vs. closures, etc.), the third mechanism makes a mess of documentation and intimidates users. Because the need for simple, clear mechanisms to control interfaces may be the most pressing problem identified in this paper, it may be worthwhile to speculate about a new mechanism.

The speculative idea follows the model of the matching statement; imagine expressing *every* capability of a program generator as an extension to the syntax of the target language. In the case of the Toolkit, for example, we might add these productions (and many more) to the grammar for C:

$\langle \textit{definition} \rangle ::= \langle \textit{definition of encoding procedure} \rangle$
$\langle \textit{definition of encoding procedure} \rangle ::=$
$\quad \langle \textit{return type} \rangle\ \langle \textit{name of procedure} \rangle\ \texttt{(}\langle \textit{arguments} \rangle\texttt{)}\ \texttt{\{}\ \langle \textit{body} \rangle\ \texttt{\}}$

$\langle \textit{structure member} \rangle ::= \langle \textit{definition of encoding procedure as function pointer} \rangle$
$\langle \textit{definition of encoding procedure as function pointer} \rangle ::= \ldots$

After extending the grammar in this way, we could specify the structure of generated code by writing a *collection of sentential forms* in the extended programming language. Instead of using a command line such as

```
tools -indirect sparcbin:sparc -encoder sparcindir ...
```

a user might write the following sentential form:

```
sparcindir.h:
  struct sparc {
    ⟨definitions of encoding procedures as function pointers⟩
  };
  extern struct sparc *sparcbin;
```

As another example, a user of `asdlGen` might use a sentential form to control the names given to types in generated code. We might express the default naming scheme by writing

$\langle$*name of generated type*$\rangle$ ::= $\langle$*name of ASDL module*$\rangle$_$\langle$*name of ASDL type*$\rangle$`_ty`

A user who had no fear of name collisions and wanted shorter, more idiomatic names could drop the prefix and suffix by writing

$\langle$*name of generated type*$\rangle$ ::= $\langle$*name of ASDL type*$\rangle$

Although a sentential form is more verbose than a command-line option, it is easier for a user to tell what generated code is specified by a sentential form, because the sentential form makes the structure of the generated code obvious in a way that keywords such as `-indirect` and `-encoder` do not. Because we expect users to write these specifications rarely, we value clarity over brevity. Using sentential forms would enable us to exploit a user's ability to recognize an extended version of his or her favorite implementation language, and we can expect a typical user's "recognition vocabulary" to be large.

If a user has difficulty producing new sentential forms, we can provide a language-based editor. Such an editor is an excellent tool for creating phrases in an unfamiliar language, essentially by direct manipulation. Direct manipulation works well for novice users, and because a program generator is not likely to be needed more than once for any particular application, *every* user is a novice user.

Of course, these claims are speculative. The idea of using extended sentential forms for program generation would need considerable development before it could even be evaluated, let alone deployed.

5.2 Lessons for prettyprinting

Prettyprinting engines are well studied; optimal methods that use dynamic programming have long been known (Oppen 1980), and the problem is a favorite of functional programmers (Hughes 1995; Wadler 1999). The maturity of this technology might lure an implementor into thinking it is easy to apply, but in practice there seems to be no well-established body of knowledge on how to exploit prettyprinting engines to produce readable, idiomatic concrete syntax for commonly used programming languages. I have found little discussion of techniques or alternatives, although Baecker and Marcus (1990) do survey previous work and present suggestions for C, and Blaschek and Sametinger (1989) present a prettyprinter designed to be adapted to different conventions.

The difficulty of producing readable output can be exacerbated by idiosyncratic characteristics of automatically generated code. For example, an RTL-creation procedure, such as `fdivd` on pages 8–9, requires function calls that are nested more deeply than calls C programmers typically write by hand, and the prettyprinted code can be very difficult to read.

Because prettyprinting is so difficult, it makes sense to have a single back-end component that concentrates on prettyprinting and ignores other issues. The input to such a component would be a form that represents the abstract syntax of the target language. Such a component and intermediate form would be highly likely to be reusable. There may also be other advantages; for example, type checking in the program generator might prevent one from generating syntatically incorrect code, as Thiemann (2000) demonstrates for HTML.

Some existing prettyprinting engines are limited in ways that preclude their use for program generation. For example, the original Toolkit's prettyprinter required a special hack to enable it to insert `#line` in column 1. One reason the original Toolkit does not emit macros for encoding is that its prettyprinter cannot easily escape internal newlines with backslashes.

5.3 Lessons for intermediate representations

This paper concludes with lessons learned about intermediate forms for reusable program generators. Although the answers are not definitive, these lessons address the following questions:

- Should an intermediate form represent the intersection of the target languages and interfaces, the union of the target languages and interfaces, or some abstraction of the target languages and interfaces? Or perhaps a combination?
- How many layers of intermediate representation should be inserted between the program-generation algorithm and the eventual emission of code?
- What intermediate representations can usefully be shared among program generators? What data or code in the program generator should be parameterized by the client's needs? How should these parameters be expressed?
- What are the relative merits of templates and trees as a basis for an intermediate representation?
- When trees are used, should there be many kinds of nodes or only a few?

Union languages versus intersection languages

The Toolkit's intermediate form is a "union language;" it contains all the features that might be used from each target language. This design makes it easy for the front end to choose an idiomatic representation of generated code, but this benefit comes at a significant cost: both the intermediate form and the renderers are large and complex. Another cost is that a transducer may be needed to match the intermediate form to the target language. For example, the Toolkit's intermediate representation includes record-valued expressions and safe unions. Because these constructs cannot be expressed in C, they must be rewritten in terms of record variables and unsafe unions; the transducer that does this rewriting is 300 lines of ML. To emit ML, the Toolkit uses a different transducer, which helps convert case statements to pattern matches.

An intersection language has the advantage of simplicity. Because an intersection language is a subset of every target language, it is easy to write emitters. If the source language is an intersection language, as is ASDL, it is easy for users to predict the results of program generation. But an intersection language may make it harder to generate idiomatic, readable code; natural idioms in the target language may not be expressible directly in the intersection language. This situation also requires a transducer; for example, the translator component of `asdlGen`'s back end may be considered a transducer. Another problem with an intersection language as source language is that the restrictions it imposes may annoy users. For example, one of my biggest frustrations in using ASDL was not having Booleans or characters built in; other users report similar frustrations.[9] While the author of a program generator may use multiple-language support as an argument for restriction, any single user may be interested in just one target language and may want the full capability of that target language.

In one small area, I think the tension between union languages and intersection languages can be resolved in favor of union languages. Polymorphic types are just too useful to leave out of a program generator, even if the target language is a monomorphic language such as C. For example, ASDL supports fully polymorphic list and option type constructors; the key is that `asdlGen` emits code only for fully instantiated monotypes. This policy could easily be extended to ASDL's algebraic types, and for languages that support polymorphic types, the restriction to monomorphism could be relaxed. As another example, λ-RTL supports a "bit vector" type constructor that is polymorphic in its width (the number of bits). The λ-RTL translator enforces a similar restriction; in the semantics of any particular machine instruction, all the widths must be known, i.e., the types must be monomorphic.

Experience with both the Toolkit and with `asdlGen` suggests that transducers may be a fact of life when supporting multiple target languages; neither an intersection language nor a union language can be a good fit with all targets. It may be possible to make a virtue of necessity by using transducers to get many of the benefits of both union and intersection languages. In a future design, I would try to use a small intersection language inside the implementation, while using a transducer to reduce the mismatch between the intersection language and the target language of the user's choice.

[9] Private communication from Dan Wang, May 2000.

Many forms versus few forms

The original Toolkit tried to minimize the number of different intermediate representations used in the program generators. A single representation of expressions is used to solve equations, to simplify expressions, and to emit code. That representation is mapped directly to strings with prettyprinting escapes; the only language-specific intermediate representation is essentially concrete syntax. These choices made it difficult to adjust the output of the program generator, and they make it unlikely that any of the code can be used in another program generator. Most of the choices have been repeated in the second implementation, with similar results.

Learning from **asdlGen**, in a future design I would use an explicit, abstract representation of each target language, and I would create prettyprinters to work with those representations, not with my general-purpose intermediate form. No matter what the merits of the rest of the design, the prettyprinters would be almost guaranteed to be reusable.

Mixing general-purpose and domain-specific constructs

Section 4.3, which describes the Toolkit's intermediate form, identifies many general-purpose language constructs which might profitably be used in other program generators. Unfortunately, the Toolkit's implementation carelessly mixes these constructs with special-purpose constructs. In a future design, I would aim for a single, extensible representation for general-purpose constructs, which would then be extended with special-purpose constructs. Not only the intermediate-tree datatype but also the language-dependent emitters should be extensible. Two-level types, as described by Steele (1994) and Sheard (2001), might be a good mechanism for extensibility.

Templates versus trees

In rewriting the Toolkit, I stopped using code templates because I thought a tree representation would be more abstract, language-independent, and easier to port. I expected easier porting because the effort required to build a new emitter should be proportional to the number of constructs in the abstract tree representation, whereas the effort required to rewrite all the templates is proportional to the number of *uses* of those constructs. Trees also reduce the cost of changing a program generator, because a change in the generated code can be effected by changing only *one* tree, whereas such a change might require changing multiple templates: at least one per target language supported.

I overlooked two problems with abstract tree representations. The first problem is that in the implementation of a program generator, code that emits code is *much* harder to read, understand, and change than are code templates. Sometimes this problem can be mitigated by crafting syntax of the emitting code to reflect the syntax of the emitted code. For example, it is possible to define a binary function in ML that builds syntax for a left-shift expression, to name that function `/<</`, and to make that name an infix operator. There are frustrating limitations to this technique; for example, it is impossible to model *all* of C's infix operators in this

way, because C's operators have fourteen levels of precedence, and ML permits infix operators to be assigned at most ten distinct levels of precedence.[10]

The second problem I overlooked is that it is hard to design a tree abstraction that works well for multiple target languages. As discussed above, it seems impossible to avoid transducers in an implementation that supports multiple languages.

In a future design, I would try harder to find an implementation technique that offered more of the clarity and simplicity of templates, even if the number of lines of code to be ported were greater. Inventing such a technique might be an interesting problem for those interested in the design of functional languages. Some work in this area has already been done.

- Standard ML of New Jersey provides a mechanism for quotation and antiquotation, which makes it possible to use Standard ML as a metalanguage for any object language. The work may be a step in a useful direction, but it suffers from several problems: syntax in the object language must be represented as strings in the metalanguage; ML values inserted into an object-language term must all have the same type; unspecified "tricks" are needed to write object-language parsers; and terms written using the mechanism are not noticeably more readable than terms written using straight ML. It's unsurprising that this mechanism is not widely used or imitated.
- MetaML uses ML as both metalanguage and object language (Sheard, Benaissa, and Martel 2000). It enables experiments in program generation, type systems, type-directed staging, etc. MetaML can infer and check types of terms in the object language, an appealing feature that is well beyond the capability of any program generator discussed in this paper. It is too early to say what ideas from MetaML might be generalizable to a setting with more than one object language.

Rich trees versus spartan trees

If a program generator represents an intermediate form as a tree, it needs to walk that tree both in ways that are unaware of the semantics of particular operators (e.g., to find free variables) and in ways that are aware of the semantics of particular operators (e.g., to perform algebraic simplification). In today's programming languages, it is difficult to choose a representation that makes both kinds of tree walks simple. There appear to be two alternatives: a "rich" representation that uses a different kind of node for each operator and a "spartan" representation that uses a single "apply" node for all operators. For example, to implement left shift, a rich representation might define a special `LSHIFT` node of type `exp * exp -> exp`, where a spartan representation might simply define a left-shift operator and use it with an `APPLY` node of type `operator * exp list -> exp`.

It is unclear which representation would better support an infrastructure for building reusable program generators. The rich representation makes it easier to

[10] An easy fix would be to extend ML so that the precedence of an infix is specified using a real-number literal. λ-RTL uses such a scheme, and the implementation is straightforward.

match particular operators but harder to walk trees; the spartan representation makes it easier to walk trees but harder to match particular operators. I have used both alternatives, and in both cases I have felt it necessary to mitigate the problems by using a special-purpose program generator to generate code for the more awkward tree operations,[11] but the results are unsatisfying.

I don't see how to resolve the conflict without some form of language extension. Perhaps an extension to pattern matching could make code clearer when using a spartan representation, or maybe some kind of generic or polytypic programming could be used to write tree walks over a rich representation. This problem warrants further investigation.

5.4 Conclusion

Reusability of a program generator is likely to be determined by the care with which its design addresses the pragmatic aspects of program generation, not only the domain-specific aspects. The most important pragmatic aspects appear to be support for multiple languages and specification of interfaces to and internals of generated code. The experience described in this paper has uncovered a few rudimentary techniques that may help, but substantial work remains to be done before we understand how to make program generators reusable.

Acknowledgments

Eric Eide's enthusiasm and thoughtful review spurred me to greater efforts. Dave Hanson's analysis of ASDL helped me clarify my own thinking. The paper by Odersky and Wadler (1997) provided the terms *heterogeneous* and *homogeneous.*

The comments of JFP's anonymous referees were unusually helpful. In particular, reviewer C suggested a restructuring that improved the paper immeasurably. An earlier version of this paper appeared the 2000 workshop on *Semantics, Applications, and Implementation of Program Generation*; comments from Dan Wang, from the program committee, and especially from reviewer 3 were very helpful. Thanks also to Tim Sheard for his stimulating questions during the workshop and to Dan Wang for our many vigorous discussions about ASDL.

My work on the Toolkit and on other program generators has been supported by a Fannie and John Hertz Fellowship, an AT&T PhD Fellowship, Bellcore, Bell Labs, DARPA Contract MDA904-97-C-0247, NSF Grants ASC-9612756 and CCR-9733974, and an Alfred P. Sloan Research Fellowship.

References

[11] Only the program generator that provides support for walking rich representations is actually implemented; when using spartan represntations, I have suffered through unreadable pattern matches.

Aho, Alfred V. and Steve C. Johnson. 1974 (June). LR parsing. *Computing Surveys*, 6(2):99–124.

Auerbach, Joshua, C. Barton, Mark Chu-Carroll, and Mukund Raghavachari. 1999 (May). Mockingbird: Flexible stub compilation from pairs of declarations. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, pages 393–402, Washington - Brussels - Tokyo.

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

Baecker, Ronald M. and Aaron Marcus. 1990. *Human Factors and Typography for More Readable Programs.* Reading, MA: Addison-Wesley.

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.

Barbacci, Mario R. and Daniel P. Siewiorek. 1982. *The Design and Analysis of Instruction Set Processors.* New York, NY: McGraw-Hill.

Batory, Don, Clay Johnson, Bob MacDonald, and Dale von Heeder. 2000 (July). Achieving extensibility through product-lines and domain-specific languages: A case study. In *Proceedings of the Sixth International Conference on Software Reuse*, pages 117–136.

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.

Blaschek, Günther and Johannes Sametinger. 1989 (July). User-adaptable prettyprinting. *Software—Practice & Experience*, 19(7):687–702.

Braun, Owen C. 1996 (May). Retargetability issues in worst-case timing analysis of embedded systems. Bachelor's thesis, Dept of Computer Science, Princeton University.

Bumbulis, Peter and Donald D. Cowan. 1993 (March). RE2C: A more versatile scanner generator. *ACM Letters on Programming Languages and Systems*, 2(4):70–84.

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

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

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

Dean, Jeffrey, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. 1996 (October). Vortex: An optimizing compiler for object-oriented languages. *OOPSLA '96 Conference Proceedings,* in *SIGPLAN Notices*, 31(10):83–100.

DeRemer, Frank and Thomas Pennello. 1982 (October). Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649.

Derman, Emanuel and Christopher Van Wyk. 1984 (December). A simple equation solver and its application to financial modelling. *Software—Practice & Experience*, 14(12):1169–1181.

Engler, Dawson R. 1996 (May). VCODE: a retargetable, extensible, very fast dynamic code generation system. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 31(5):160–170.

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

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

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.

Gibbons, Phillip B. 1987 (January). A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, SE-13(1):77–87.

Gray, Robert W. 1988 (June). γ-GLA: A generator for lexical analyzers that programmers can use. In *Proceedings of the Summer USENIX Conference*, pages 147–160, Berkeley, CA, USA.

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

Hanson, David R. 1996. *C Interfaces and Implementations.* Benjamin/Cummings.

———. 1999 (April). Early experience with ASDL in `lcc`. *Software—Practice & Experience*, 29(5):417–435. See also Technical Report MSR-TR-98-50, Microsoft Research.

Hughes, John. 1995. The design of a pretty-printing library. In Jeuring, Johan and Erik Meijer, editors, *Advanced Functional Programming*, Vol. 925 of *Lecture Notes in Computer Science.* Springer Verlag.

Hunt, Andrew and David Thomas. 1999. *The Pragmatic Programmer: From Journeyman to Master.* Reading, MA: Addison-Wesley.

Johnson, Steve C. 1975. Yacc—yet another compiler compiler. Technical Report 32, Computer Science, AT&T Bell Laboratories, Murray Hill, New Jersey.

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.

Lee, Peter and Mark Leone. 1996 (May). Optimizing ML with run-time code generation. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 31(5):137–148.

Lesk, M. E. and E. Schmidt. 1975. Lex — A lexical analyzer generator. Computer Science Technical Report 39, Bell Laboratories, Murray Hill, NJ.

Liskov, Barbara and John Guttag. 1986. *Abstraction and Specification in Program Development.* MIT Press / McGraw-Hill.

Nielsen, Jakob. 1993. *Usability Engineering.* Boston, MA: Academic Press.

Odersky, Martin and Philip Wadler. 1997. Pizza into Java: Translating theory into practice. In *Conference Record of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 146–159. ACM SIGACT and SIGPLAN, ACM Press.

Oppen, Derek C. 1980 (October). Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483.

Proebsting, Todd A. 1992 (June). Simple and efficient BURS table generation. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 27(6):331–340.

Ramsey, Norman. 1994 (January). Correctness of trap-based breakpoint implementations. In *Proceedings of the 21st ACM Symposium on the Principles of Programming Languages*, pages 15–24, Portland, OR.

———. 1996a (May). Relocating machine instructions by currying. *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 31(5):226–236.

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

———. 1998 (October). Unparsing expressions with prefix and postfix operators. *Software—Practice & Experience*, 28(12):1327–1356.

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.

———. 1999 (December). Specifying instructions' semantics using λ-RTL (interim report). See `http://www.cs.virginia.edu/zephyr/csdl/lrtlindex.html`.

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

———. 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.

Sheard, Tim. 2001 (September). Generic unification via two-level types and parameterized modules. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 86–97, New York, NY.

Sheard, Tim, Zino Benaissa, and Matthieu Martel. 2000 (February). *Introduction to Multistage Programming Using MetaML*. Revision 2. Pacific Software Research Center, Oregon Graduate Institute of Science and Technology, PO Box 91000, Portland, OR 97291-1000. See `http://www.cse.ogi.edu/PacSoft/projects/Mustang/Overview.html`.

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

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

Steele, Guy Lewis, Jr. 1994. Building interpreters by composing monads. In ACM, editor, *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 472–492, New York, NY, USA.

Thiemann, Peter. 2000 (January). Modeling HTML in Haskell. In Pontelli, Enrico and Vítor Santos Costa, editors, *Practical Aspects of Declarative Languages (PADL 2000)*, Vol. 1753 of *Lecture Notes in Computer Science*, pages 263–277. Berlin: Springer.

Wadler, Philip. 1999. A prettier printer. Unpublished note available from the author's Web site.

Waite, William M. 1986 (May). The cost of lexical analysis. *Software—Practice & Experience*, 16(5):473–488.

Wang, Daniel C., Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. 1997 (October). The Zephyr Abstract Syntax Description Language. In *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, pages 213–227, Santa Barbara, CA.

Wang, Daniel C. 1999. `AsdlGen` *Reference Manual*. May be available from `http://www.cs.princeton.edu/zephyr/ASDL`.