Pragmatic Aspects of Reusable Program Generators
Position Paper

Norman Ramsey

Division of Engineering and Applied Sciences
Harvard University
nr@eecs.harvard.edu

Abstract. When writing a program generator requires considerable intellectual effort, it is pleasant 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 uses examples from SLED, λ-RTL, and ASDL to elaborate on these questions. It is hoped that the paper will stimulate discussion and the development of better techniques. Most urgently needed is 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 may be used only once, while others are used over and over. 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 might justify such an effort.

- Generating a lexical analyzer requires algorithms that manipulate finite automata, e.g., they translate regular expressions to NFAs, convert NFAs to DFAs, and minimize states of DFAs. 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 the 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).

These 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 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.

This paper presents problems, not solutions. It reports on experience with the program generators behind the New Jersey Machine-Code Toolkit (Ramsey and Fernández 1995) and the λ-RTL translator (Ramsey and Davidson 1999), as well as the ASDL program generator (Wang et al. 1997). This experience shows the significance of the problems, even if the solutions used in the tools are often unsatisfying.

[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).

1.2 Requirements for reuse

What properties should a program generator have if it is to be used?

- The generated code should be *written in the programming language of the client's 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.* Wise programmers distrust generated code that they don't understand (Hunt and Thomas 1999, Section 35). Readable code is doubly important when an author is debugging a program generator that is new or that involves experimental algorithms.
  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 user 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 (Ramsey and Fernández 1995).
- A user may wish to *control internals* of the generated code, for example, to adjust space/time tradeoffs or other properties that affect performance.

In a single-use program generator written by its user, these properties should emerge naturally. They are much harder to achieve in a program generator intended for reuse; after all, these properties appear peripheral to a program generator's main job, which is to embody some useful algorithm or technique. The primary claim of this paper is that these pragmatic aspects of program generation are every bit as important as the central, algorithmic aspects. The primary source of evidence supporting this claim is experience with the New Jersey Machine-Code Toolkit.

2 Case Study: The New Jersey Machine-Code Toolkit

The New Jersey Machine-Code Toolkit is a program generator that helps programmers create applications that process machine code—assemblers, disassemblers, code generators, tracers, profilers, debuggers, and executable editors, for example. Writing low-level bit-fiddling code by hand is tedious and error-prone; generating this code automatically reduces retargeting effort and eliminates a significant source of errors.

Applications use the Toolkit for encoding, decoding, or both. For example, assemblers encode, disassemblers decode, and executable editors do both. 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), a binary translator (Cifuentes, van Emmerik, and Ramsey 1999), and other applications. All applications work with *streams* of instructions. Streams can take many forms; for example, a debugger can treat the text segment of a target process as an instruction stream.

2.1 Coupling generated code to applications

Decoding applications use *matching statements* to read instructions from a stream and identify them. A matching statement is like a case statement, except its alternatives are labelled with patterns that match instructions or sequences of instructions. Client programmers insert matching statements directly into hand-written source code. The Toolkit transforms matching statements into efficient decoders.

Encoding applications call C procedures generated by the toolkit. These procedures encode instructions and emit them into a stream; e.g., the SPARC call `fnegs(r2, r7)` emits the word `0x8fa000a2`. The Toolkit can also create encoding procedures that emit assembly language.

The Toolkit includes two program generators, which share infrastructure. The *translator* translates matching statements in a C or Modula-3 program into ordinary code. The *generator* generates encoding and relocation procedures in C.

Generated code couples to applications' code at both higher and lower levels of abstraction. Application code at high levels of abstraction calls generated code to encode or decode instructions. Generated code calls application code at lower levels of abstraction to manage the details of reading and writing the bytes in the *instruction stream*, which is the Toolkit's model of binary representation.

Low-level coupling: generated code to instruction streams The Toolkit dictates the semantics, but not the representation, of instruction streams. An instruction stream is like a byte stream, except that the units may be "tokens" of any size, not just 8-bit bytes. Because machine instructions don't always fit in a machine word, an instruction is a sequence of one or more tokens; for example, a SPARC instruction is always 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.

Decoders generated by the Toolkit examine tokens in instruction streams. Each decoder treats an *address* within a stream as an abstract type; application code supplies a representation, as well as code to perform three operations: add an integer to an address, fetch a token from an address, and convert an address to an integer (e.g., so a decoder can extract bits from the program counter). The Toolkit's decoders do not use linguistic abstraction mechanisms (e.g., C procedures or C preprocessor macros) to manipulate an application's instruction stream; instead, the Toolkit uses its own form of macros. Users provide "code templates" that perform the basic operations above, and the Toolkit instantiates these templates as needed.

The templates define the interface between the generated code and the lower-level application code. For example, these templates are 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)"
address to integer using "%a"
fetch any using "FetchAbs(m, %a, Type.I%w).n"
```

The instruction stream, `m`, is built directly into the fetch template.

As another example, these templates are used in a binary translator, written in C (Cifuentes, van Emmerik, and Ramsey 1999); the instruction stream is implicit in the `getDword` operation.

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

Encoding procedures generated by the Toolkit have a different low-level interface, which is a single C procedure or macro. Each encoding procedure calls that procedure to emit a token into the instruction stream; the Toolkit's user can choose the name of that procedure at program-generation time. 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.

High-level coupling: application to generated code At the high level, there is no formal interface between an application and a generated decoder. Matching statements act just like statements in the source language (C or Modula-3), and each action within a matching statement may be any statement in the source language. In effect, the Toolkit extends the source language with a new construct.

The Toolkit generates a procedural interface for access to encoders. Applications call through this interface to emit binary representations into an instruction stream. The Toolkit supports not only direct calls, but also indirect calls, which go through function pointers stored in a structure. The indirect method is the standard encoding of object-orientation in C.

The alternatives that we have implemented are not the only alternatives; Section 3.1 discusses high-level interfaces further.

2.2 History and requirements

The Toolkit evolved from single-use program generators used to help retarget Standard ML of New Jersey (Appel and MacQueen 1991) and the debugger `ldb` (Ramsey and Hanson 1992). Standard ML of New Jersey uses run-time code generation, and `ldb` uses control-flow analysis of machine instructions to implement breakpoints (Ramsey 1994). Fernández's (1995) work on link-time optimization needed a fast, link-time assembler for performance, and this need provided the motivation to merge and adapt the original generators to create the Toolkit. Because the original applications were written in Standard ML, Modula-3, and C, support for multiple languages was a goal from the beginning.

Because the work was experimental, we placed a high premium on the readability of the generated code. Readable code not only helped us debug the program generator; it also helped us understand the performance and behavior of the generated code. Reading the generated decoders was especially important, because it helped us compare them with hand-written decoders, which made it easier to develop the decoding heuristics that make generated decoders perform as well as hand-written decoders.

We were also aware early that generated code should be idiomatic, i.e., where possible it should resemble code that a fluent programmer would write by hand. For example, the Toolkit should not require its own printing abstraction in order to emit assembly language; in C, it should use `printf`, and in Modula-3, it should use writers.

The need for users to control the interfaces to the generated code and some internals of the generated code was not obvious early. It become clear only as the Toolkit acquired other clients.

3 Needs of clients

Of the requirements in Section 1.2, the most problematic have been the need to support multiple languages and the need to adjust generated code to interoperate cleanly with client code. This section illustrates some of these needs, using examples drawn from the Toolkit and from other program generators. The section then moves to more general issues of concrete types, interfaces, and languages.

3.1 Examples from the Toolkit

Needs of decoding applications The design of the Toolkit's matching statements simplifies not only the program generator, but also the coupling to client code. Because the application programmer drops matching statements inline wherever decoders are 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. The code templates make it unnecessary to decide how to package the instruction-stream abstraction, and they support multiple languages effortlessly.

These properties provide enormous flexibility while keeping the program generator simple. For example, clients can easily arrange for decoders to manipulate instruction streams without the overhead of procedure calls. No special command-line options or configuration parameters are needed, and the client need not rely on a compiler to inline procedure calls or method calls.

One penalty of the approach is that errors in matching statements or in templates can be detected only *after* the program generator runs, and some 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 statements.

Needs of encoding applications Despite the advantages we realized by making the decoder generator a preprocessor, it never occurred to us to design the encoder generator that way. In both cases we were thinking not about the problem of coupling generated code to client code (the "pragmatic aspects" that are the subject of this paper) but about the best model for the program generator itself. For decoding, we wanted "ML-like pattern matching over binary representations of machine instructions." For encoding, we wanted "something just like assembly language, but more flexible than a traditional assembler." In particular, we wanted to emit instructions by name, and to have the assembler make sure the types and values of the operands were sensible. This thinking led us to associate an encoding procedure with each machine instruction.

This model serves the task of encoding instructions: check that the operands are sensible, and produce the bits. But we failed to consider that different applications may take different views of the *interface* to encoding.

- In the *applicative view*, instructions are abstractions, which are built up 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, i.e., emission of the binary representation of the instruction. This view is congenial to traditional assemblers.
- Our original view is 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.

[2] For those unfamiliar with CISC assembly languages, an effective address is a fragment of a machine instruction that identifies the location of an operand or result. In typical instruction sets, this location may refer either to a machine register or to a location in memory.

Where	*Who decides*	*What is decided*	*Possible language constructs*
Above	application	what to encode when	procedures, objects, (macros)
Gen. code	generator	encoding logic	
Below	application	how to emit bits	procedures, macros

Table 1. Interface layers for encoding applications

- 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 semantics of the interfaces is not all that can vary; the language constructs used to express the interfaces can also vary, as Table 1 illustrates. For the interface from above, the Toolkit enables the user to control the names of the generated procedures, and whether the procedures are called at the top level or indirectly through function pointers. The interface from below is simpler; the generated code needs to call only one emission procedure or macro.

Do these interfaces matter? Unfortunately, yes. Applications that generate code dynamically, such as DyC (Auslander et al. 1996), Fabius (Lee and Leone 1996), and VCODE (Engler 1996), all want the absolute best in performance. Overhead for indirect calls would be unacceptable. In the case of VCODE, even procedure-call overhead is unacceptable; VCODE cannot be implemented using the Toolkit because it requires macros.[3] More traditional applications, like static compilers, may 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.

We and our users paid for our oversight. When the Toolkit was first distributed, each new user asked for a new variation. We implemented several, but each addition increased the complexity not only of the Toolkit's implementation, but also the complexity of the interface used to control the program generator. The Toolkit uses command-line options to control variations (applicative or imperative, direct or indirect) and naming (of procedures, of types, and of structures). To achieve even the incomplete control provided by the Toolkit, there are 9 options that affect the interfaces to encoding procedures. Another 3 affect internals (e.g., logging), and perhaps a half dozen affect the closely related relocation procedures.

This situation is intolerable. The plethora of possibilities baffles even the Toolkit's authors; never mind what it does to novices. Clearly, some alternative is needed. I have considered a configuration language, but have not been enthusiastic; why should a big set of configuration variables be significantly better than a big set of command-line options? What is surprising is that the decoder generator avoids these problems completely, by requiring the client to provide

[3] Private communication from Dawson Engler, 1995.

source code both above (surrounding) matching statements and below matching statements (the code templates). What I really desire for the encoder generator is some sort of "program generation by example" that would banish command-line options and configuration variables, enabling users to say "generate code that looks like *this*." I wish I knew what such a facility might look like.

3.2 Examples from ASDL

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`, emits code to create trees, pickle them (write them to files), and unpickle them (read them from files). Because the pickle format is independent of the programming language in which the trees are created, written, and read, components implemented in a variety of programming languages can exchange trees. `AsdlGen` 2.0 supports C, C++, Haskell, Icon, Java, Objective Caml, and Standard ML.

ASDL describes trees using a restricted form of algebraic datatypes, derived from types found in ML and Haskell. The value of the program generator is twofold:

- It generates an applicative interface for the creation of trees. While tree creation is trivial in languages like ML and Haskell, languages like C require explicit, separate allocation and initialization. The program generator supplies code that would be tedious to write by hand.
- It generates picklers and unpicklers for serialization, which would be very tedious to write by hand. Even better, the serialization code guarantees interoperability.

`AsdlGen` does these jobs well, but like the Toolkit, it does not work so well when it comes to the interfaces between generated code and client code.

Basic structure of interfaces The original version of `asdlGen` interleaved type definitions, constructors, and serialization code in a single interface. This choice violated the requirement that generated code be idiomatic and easy to understand. While any module using the generated code will need access to the type definition, most will use either constructors, picklers, or unpicklers, 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. Worse, it creates a configuration problem: code generated with `asdlGen` can't be compiled without an installation of `asdlGen` that provides serialization libraries.

A later version of `asdlGen` split the output into two interfaces: one for definition and construction, one for serialization. This split simplified the migration of existing projects to ASDL, since it become possible for an ASDL expert to generate definitions of types and constructors, then ship the generated code to another site to be used.

Interfaces and modularity One of the most significant defects in ASDL is that it is difficult to define trees that contain subtrees of types defined in different modules. For example, if a programmer is designing a compiler, he might want to define statements and expressions in different modules. Because statements may contain expressions, this structure is difficult to achieve using ASDL, even if both expressions and statements are defined in ASDL. The situation worsens if a programmer wants to define some types using hand-written code, e.g., to exploit features that are available in the target implementation language but not in ASDL. Code generated by `asdlGen` cannot easily refer to such types, even when there is hand-written serialization code available to support them.

ASDL's "view" mechanism partly addresses both problems. In the main part of the ASDL description, the programmer must pretend that an abstract type is some concrete type, e.g., a string. The "view" mechanism, which works by adding fragments of hand-written code to the ASDL description, makes it possible to override the pretend type with the actual type, and to provide conversion routines between the pretend type and the actual type. It is not possible to provide picklers and unpicklers directly, nor is it possible to assert that values of the pretend type will never be serialized. The program generator would benefit from a cleaner way to interface to types for which the definitions, constructors, picklers, and unpicklers were defined by the programmer in ordinary source files.

Abstract types and existing trees: unforeseen applications and effects One way to view an ASDL pickle is that it says which constructors should be called, in what order, to reproduce a tree. In this view, it should not matter whether the constructors build `asdlGen`'s concrete representation of trees, or whether they build a value of some abstract type. According to the algebraic approach to data abstraction, the concrete representation should be irrelevant; there is an obvious one-to-one mapping between trees of some concrete algebraic data type and a corresponding abstraction (Liskov and Guttag 1986). Unfortunately, the designers of `asdlGen` overlooked this possibility; generated picklers and unpicklers cannot couple to abstractions, but only to `asdlGen`'s concrete representation. The only way to pickle and unpickle a value of an abstract type is to write "transducers" by hand. For example, a generated unpickler produces ASDL's concrete representation; an unpickling transducer walks that representation and produces a value of an abstract type.

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.

Simon Peyton Jones and I have found that transducers are necessary even when designing a new compiler from scratch. Because we want to couple a back end to front ends written in different languages, we attempted to use ASDL to describe an intermediate form. We do not, however, want the representation defined by ASDL—only the abstract interface. Our internal representation includes private annotations: sets of registers read and written, sets of source-language variables live at a control point, lists of registers whose last use is at a given assignment ("dead lists"), etc. Such annotations, which are likely to change as the compiler evolves, should not be visible from the external interface, 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, but from an accident of `asdlGen`'s design: generated serialization code can be coupled only to ASDL's concrete representation, not to an abstraction. `AsdlGen` could easily 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. Although it is not immediately obvious how to encode such parameterization in C, many object-oriented and functional languages provide natural, idiomatic parameterization mechanisms (e.g., classes, functors, generics, and templates). The flaw in the program generator lies not in the fundamental model or the underlying algorithms, but in the limitations on the possible interfaces to the generated code.

3.3 Concrete types, polymorphic and otherwise

Both the Toolkit and `asdlgen` share with parser generators and with other program generators the need to create sequences of things. In some target languages, like ML and Haskell, polymorphic list types provide natural representations. 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, or 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).

Sequences offer choices not only of representation, but also of type. *Heterogenous* generation creates a specialized sequence type for each different kind of sequence. This choice preserves type safety at the cost of code bloat. *Homogeneous* generation uses a generic sequence type that contains something like C's `void *` or Java's `Object`. This choice abandons type safety, but does not duplicate code. Again, no single choice is always suitable; for example, when translating polymorphic types into Java, the Pizza translator uses both alternatives (Odersky and Wadler 1997).

Even simple, monomorphic types can provide pitfalls for the unwary 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 C 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?

In sum, different clients need different representations, 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.

3.4 Interfacing to abstractions

Once the author of a program generator recognizes the need to couple generated code to hand-written abstractions, the next question is what language constructs should be used at the interface. In object-oriented languages, objects with methods are natural. In functional languages, functions and closures may be more natural. Some traditional imperative languages, as well as many functional and object-oriented languages, provide modules, which may be more better representations of complex abstractions. In C, an abstraction can be represented by a group of functions, plus a pointer of unknown type (`void *`). The pointer, which corresponds to the environment pointer used to form closures in functional languages, is passed as an additional argument to each function. Using `void *` to turn a function pointer into a closure is a bit of a trick, but it is one that is often used to help make C code reusable (Hanson 1996).

The proper representation of an abstraction is influenced not only by the features available in the target language, but also by the idioms commonly used in that language. As an example, consider the problem of generating encoding procedures that print assembly language.

- In C, the use of `void *` can be made to fit the interfaces defined by `fprintf` and `sprintf`.

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

  The variable `asmprintf` can point to `fprintf` or to `sprintf`, as well as to other functions. To avoid passing extra arguments to encoding procedures, `asmprintf` and `asmprintfd` are global variables. This choice is not just convenient; it enables the encoding procedures for binary and assembly language to have identical interfaces.
- In ML, one might expect to use a closure to print assembly language, but in practice, this is not idiomatic—most ML programmers would find it more natural to return a string, or perhaps a list or tree of strings.
- In Modula-3, one might expect to use an object with a print method, but in practice, Modula-3 programmers use a "writer" abstraction, making calls through the `Wr` interface.

The point of these examples is that language paradigm alone is not enough to determine the most idiomatic representation of abstractions. Two semantically similar abstractions might have different idiomatic representations, even in the same target language. Producing idiomatic code is difficult for the program generator. In `asdlGen`, for example, one of the hardest problems has been to hide differences between target-language idioms, even for a problem as seemingly straightfoward as printing.[4]

These details 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 exploit the rich support for writers found in the Modula-3 library. These kinds of requirements seem to call for very fine control over program generation.

4 Implementations

No program generator I have written or used does a great job providing multiple interfaces to generated code, in multiple languages. Still, my attempts have identified some questions for implementors to consider. Many of these questions revolve around intermediate representations that might be used inside a program generator.

- How many layers of intermediate representation should be inserted between the program-generation algorithm and the eventual emission of code?
- What should the intermediate representations look like? Should they 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?
- What intermediate representations can usefully be shared to generate code in different implementation languages?
- What intermediate representations can usefully be shared among program generators?

The Toolkit's implementation may suggest some answers, as discussed below.

The other important questions involve controlling interfaces to generated code. What in the program generator needs to be parameterized by the client's needs? How should these parameters be expressed? How can we control the complexity of a program generator that serves diverse clients?

4.1 Implementation of the Toolkit

The Toolkit has two implementations. The original, official implementation was written in Icon (Griswold and Griswold 1996). We chose Icon because it was an excellent match for the central problems of our original decoder generator; Icon's evaluation model made it easy to write backtracking, heuristic searches for good decoders, and Icon's string facilities made it easy to write preprocessors. As we added support for multiple languages and for alternative interfaces, and

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

as the program generators grew more complex, Icon's lack of static typing and lack of modules made the implementation more and more unwieldy, and finally 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: write new, clean implementations of the algorithms, so we could add improvements, and 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). I have been disappointed in the results, and I would not recommend that anyone use either my Icon code or my ML code. The rest of this section explains why.

4.2 The Icon toolkit: trees and templates

Because Icon is a dynamically typed language, boundaries between intermediate representations are not precise or formal, but they can still be identified.

The program generator reads a machine description and produces equations relating abstract machine instructions and binary representations. The encoder generator uses an equation solver (Ramsey 1996) 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. Both the inputs to and outputs from the solver are expressed in a simple expression language. Most expressions are represented as trees, but to simplify the solver, linear combinations are represented as tables in which the keys are expressions and the values are coefficients. The equation solver is critical to the program generator's job, but it was a mistake to let the needs of the solver warp the representation of expressions used throughout the entire generator.

The program generators use the solver's expressions to create statements. The encoder generator uses relatively few statements: sequences, conditionals (for conditional assembly), case statements (for analyzing abstract effective addresses), and token emission. There are also statements that announce errors and emit relocation information. The decoder generator uses a greater variety 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 statements would be useful in a general-purpose infrastructure for program generation; some would not. Identifying a good set of statements 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.

The next layer of program generator, like the solver, is shared by the encoder generator and decoder generator. It converts expressions, statements, and directives into a "prettyprinting stream," which is represented as a string containing target-language code and embedded escape sequences. The escape sequences direct the insertion of indentation and line breaks; the interface is based on Pugh

```
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. 1. Code that instantiates an encoding-procedure template

and Sinofsky (1987). The conversion happens in two stages. First, trees (and tables) are converted to prettyprinting streams. Second, prettyprinting streams are inserted into code templates, which can be thought of as prettyprinting streams with "holes." Both the encoder generator and the decoder generator convert trees, but only the encoder generator uses templates. The module that converts trees to C is 330 lines of Icon; the converter to Modula-3 is 243 lines. The encoder generator uses about 20 templates, which are 1–5 lines of code each.

Here, for example, is the template for an applicative encoding procedure, which returns a value of an abstract type representing an "instance" of an instruction.

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

The `%` signs mark holes that are to be instantiated. The marks `$t` (tab) and `$b` (backtab) are escape sequences that control prettyprinting. Figure 1 shows the code that uses the template to emit an encoding procedure. The parameter `cons` contains information about the constructor being emitted. Each assignment to an instance variable is created by instantiating the `instance-assignment.t` template, 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. The function `Cnoreserve` mangles the name of the instruction to insure that it that 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 made `static`, so it won't be visible outside the generated compilation unit. Otherwise, it is exported.

The final step in program generation is to run the prettyprinter.

4.3 The ML Toolkit: a more abstract infrastructure

Rewriting the Toolkit in ML made it possible to separate intermediate forms explicitly, and to enforce that separation using ML's type and modules systems. Out of desire to enforce the separation, I abandoned templates, since all templates would have the same type.[5]

The ML implementation introduced an explicit representation of types in generated code. The representation includes all the types I could think of using in any target language: integers (signed and unsigned, and of arbitrary widths), Booleans, strings, characters, records, arrays, both safe and unsafe unions, functions, pointers, named abstract types, instances of machine instructions or effective addresses, a unit type (equivalent to C's `void`), and objects with inheritance. Integer types can be made "relocatable;" values of relocatable types have later binding times than values of ordinary integer types. By including both general-purpose types and Toolkit-specific types, I prevented intermediate forms from proliferating, but I also made the infrastructure harder to reuse. Were I to repeat the experiment, I might try to separate these two layers, but avoid duplication, by using some form of polymorphic extension, analagous to the "tower" of term representations used in Steele (1994).

As does the Icon implementation, the ML implementation uses a unique constructor for each operator in the term language. For example, left shift is represented not by a general "apply" node, but by a special `SHIFT` node. This representation made it easier to write the equation solver and algebraic simplifier, but it makes tree walking tedious. Accordingly, tree-walking functions are generated automatically by another single-use program generator. This generator also creates the type definitions for expressions, and functions that type-check expressions, rewrite trees, substitute for identifiers, and convert trees to LISP-like symbolic expressions. Unlike the Icon representation, the ML representation 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 requires about 30 lines of code to convert back and forth.

Expressions are embedded in statements, which also represent a union of target-language facilities. For example, they are translatable into C statements or ML expressions. They include general-purpose assignments, conditionals, case statements, block comments and commented statements, nested blocks with local variables, return statements, and an empty statement. They also include many special-purpose statements that are useful only in the context of the Toolkit, include statements that emit a token, discriminate on an abstract instruction, branch to an arm of a matching statement, etc.

Statements are included in procedure definitions. The program-generation abstraction can declare and define not only procedures, but also types, variables, constants, and exceptions. Declarations are collected into *interfaces*; definitions are collected into *implementations*, which import and export interfaces.

[5] In the ML implementation, the type of a prettyprinting stream is `PP.pretty`, and the type of a template would be `(string * PP.pretty) list -> PP.pretty`.

Both interfaces and implementations can be parameterized. The interface abstraction has special-purpose 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 emitters for C and for ML are about 600 and 800 lines, respectively, or about double the size of the emitters 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.

5 Conclusions and lessons learned

Authors of program generators should explore not only the design space of possible generated code, but the design space of possible interfaces to that code. It is too easy to make mistakes if one does not know what interfaces clients will need. 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 draw a few conclusions about implementations.

5.1 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 does not appear to be a well-established body of knowledge on how to exploit prettyprinting engines to produce readable, idiomatic results. I have found little discussion of techniques or alternatives, but Baecker and Marcus (1990) does survey previous work and present suggestions for C, and Blaschek and Sametinger (1989) presents 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, the λ-RTL translator uses the Toolkit's infrastructure to generate RTL-creation procedures. These procedures are like encoding procedures, but they emit the register-transfer lists (RTLs) that are used as intermediate code in the optimizer vpo (Benitez and Davidson 1988). Creating these RTLs requires function calls that are nested more deeply than calls C programmers typically write by hand, and the prettyprinted code is very difficult to read.

In the interests of parsimony, the original Toolkit translates directly from its intermediate form to target-language prettyprinting streams. This design has made it surprisingly difficult to get code that is well prettyprinted; for example, the output from the Toolkit still contains occasional unwanted newlines.

A better design would separate the problem of translating into the target language from the problem of prettyprinting the target language. For example, one might introduce yet another intermediate form, to represent code in the target programming language. Dan Wang, implementor of `asdlGen`, reports satisfactory experience with this sort of design.[6] The ML implementation of the Toolkit takes a few steps in this direction; it has special intermediate forms to represent C types and ML pattern matches, for example.

Although adding an intermediate form for each target language might seem superficially unattractive, it does offer advantages. For example, type checking in the program generator might prevent one from generating syntatically incorrect code, as Thiemann (2000) demonstrates for HTML. Another advantage is the possibility of reusing the intermediate form and its prettyprinter in multiple program generators.

Some existing prettyprinting engines are limited, so program generators may not be able to use them without modification. 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.

5.2 Intermediate representations

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, with no abstract representation of the target programming language. 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.

The Toolkit's intermediate representations are "union languages," containing all the features that might be used from each target language. This design simplifies high levels of the program generator, but it can make the emitters larger and more complex. For example, the 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 rewriter is another 300 lines of ML.

Intersection languages have the advantage of simplicity. Because the intersection language is a subset of every target language (for generated code), 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 intersection languages may make it harder to generate idiomatic, readable code; natural idioms in the target language may not be expressible directly in the intersection language. Also, users may chafe at restrictions imposed by the use of

[6] Private communication, 1998.

intersection languages; after all, while the author of a program generator may want to support multiple languages, any single user may be interested in just one language. For example, one of my biggest frustrations in using ASDL to create ML code was not having Booleans or characters built in. All I wanted was to save and restore my compiler's intermediate codes, and I was furious to have to write crippled ML code just because (in my perception) C doesn't have Booleans. Other users report similar frustrations.[7]

In the ML implementation of the Toolkit, I tried to build a reusable infrastructure for program generation. Although I successfully used the infrastructure a second time, I consider it a failed experiment. It is always difficult to learn from negative results, but here are the most pressing concerns I would try to address in another attempt.

- The intermediate form tries to generalize over many target languages. This choice makes the code-generation interfaces large and complex, without clear benefit.
- The intermediate form intermixes general-purpose and special-purpose constructs promiscuously. There should be a single, extensible representation for general-purpose constructs, which would then be extended with special-purpose constructs. Therefore not only should the intermediate-tree datatype be extensible; all the language-dependent emitters should also be extensible.
- A program generator needs to walk intermediate trees 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. I tried to address this problem by using program generation for one set of tree walks (the set that is unaware of operators' semantics), but the results are baroque. This problem warrants further investigation.
- 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. At least the prettyprinters would almost certainly be reused.
- Both the ML and C emitters contain special-purpose transducers. Since these appear to be needed even when using a big union language, I would be tempted to try a small intersection language.
- I abandoned code templates because I thought a fully abstract tree representation would be more principled and easier to port. But in the implementation of a program generator, templates are *much* easier to read, understand, and change than is code that emits code. And it is hard to design an abstraction that really works for multiple target languages. So I would try to find a solution that offered more of the clarity and simplicity of templates, even if the porting cost were greater.

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

5.3 Controlling interfaces

Instantiating code templates by textual substitution has no semantic basis, making it almost impossible to detect and report errors in terms of the original sources. But templates not only simplify life for the implementor; they can also make things easier for the clients. Because it uses code templates, the interface to the Toolkit's decoder generator is very simple. In combination with the matching statements, the templates make the decoder generator easy and pleasant to use.

The contrast with the Toolkit's encoder generator is shocking. The problem is how to control all the potential variations: applicative vs. imperative, direct vs. indirect, objects vs. closures, etc. Command-line options, and probably also configuration variables, are too complex. The complexity makes a mess of documentation and intimidates users. A fixed interface is no better; for example, the fixed interfaces of code generated by Yacc and Lex make it hard to have multiple lexers or parsers in one application. The need for simple, clear ways of controlling interfaces to generated code may be the most pressing problem identified in this paper.

Acknowledgments

In this paper, the first-person plural describes joint work with Mary Fernández, co-creator of the New Jersey Machine-Code Toolkit. Thanks to Dan Wang for comments on the paper and for many vigorous discussions about ASDL. Thanks to Dave Hanson for his analysis of ASDL, and for suggesting using distinct codes for all enumerations. I borrowed the terms *heterogeneous* and *homogeneous* from Odersky and Wadler (1997). Comments from the program committee, and especially the many questions asked by reviewer 3, have helped me improve the paper. 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, and NSF Grants ASC-9612756 and CCR-9733974.

References

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

Appel, Andrew W. and David B. MacQueen. 1991 (August). Standard ML of New Jersey. In Wirsing, Martin, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York.

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.

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.

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.

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, J. and E. Meijer, editors, *Advanced Functional Programming*, Vol. 925 of *LNCS*. 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.

Pugh, William W. and Steven J. Sinofsky. 1987 (January). A new language-independent prettyprinting algorithm. Technical Report TR 87-808, Cornell University.

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.

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

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

———. 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.

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.

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

Steele, Jr., Guy L. 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 *LNCS*, 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.