Building program generators the easy way
(Extended Abstract)

Samuel Kamin*
Computer Science Dept.
University of Illinois at Urbana-Champaign
s-kamin@uiuc.edu

Abstract

Program generators are languages whose programs produce programs in other languages. We show how program generators can be built easily as sets of definitions in a functional language. The method we present has the potential to be a useful application of functional languages in software development projects based on conventional languages. We illustrate the method with two examples: a top-down parser generator; and a version of the "Message Specification Language," developed at Oregon Graduate Institute. In the latter case, we are able to implement substantially the same language in 100 lines of Standard ML code, with no additional tools or optimizing compilers needed.

Keywords: program generators, functional programming, domain-specific languages

1 Introduction

Functional programming languages might gain wider acceptance in the software engineering world if it were easier to incorporate them into on-going software projects, almost all of which are based on conventional general-purpose languages. This incorporation is impeded by both technical and non-technical factors. The long and short of it is that project managers are extremely reluctant to incorporate functional language code in delivered products.

On the other hand, programming tools used in the software engineering process — editors and other string-processing languages; databases; document processors — include a variety of languages. Since these tools are not part of the delivered product, they are accepted far more readily. One kind of tool is a *program generator*, which accepts a specification in a domain-specific language and produces code in the project's central programming language.

This paper presents a new way of producing program generators: as sets of definitions in a functional language (Standard ML, though any functional language would do). This method has two major advantages over other techniques of producing program generators.

First, it is much simpler. Our main example in this paper is a version of the Message Specification Language described in [3, 15]. As described there, the language development involves an elaborate set of tools, including a very aggressive optimizing compiler for a functional language, which must translate functional code into conventional language (ADA) code. Our version is completely defined in about 100 lines of Standard ML, which are given in this paper. The code produced by our program generator is completely specified in those 100 lines, and does not depend in any way on the vagaries of the ML compiler.

The second advantage of our method is that the language we obtain is an extension of ML and therefore inherits all the features of ML. We show in our examples how this power is useful in a program generator. Almost all program generators in use are strictly first-order,

*Supported by NSF Contract CCR 93–03043.

but there is no reason that they might not benefit from higher-order constructs (as might also be said for many other special-purpose languages).

This paper contains two examples of program generator construction. We begin, in section 2, with a general description of the process of building a program generator. Section 3 gives a simple example: a program generator for top-down parsers. Section 4 gives our major example in this paper, a version of the Message Specification Language [3, 15] implemented using our method. We end with some conclusions about the power and limitations of the method.

1.1 Related work

Program generators are very common, but program generating languages do not always seem to be considered as full-fledged languages, with data types, control structures, and so on. Examples include Gelernter's program builders [1], and Waters's KBEmacs [16]. Each provides a fixed set of program-building operations, but there is no direct manipulation of programs, nor is it easy for users to define their own operations.

Work at OGI [3] (on which section 4 of this paper is based) and at ISI [2] is specifically aimed at the development of program generators. In both cases, the specification is translated to a very-high-level language, from which an imperative language program is extracted. Our approach is more direct: the program-generating language provides operations that manipulate *programs*, not *specifications*. One contribution of this paper is to show that such a non-abstract approach can yield quite satisfactory results in terms of usability of the resulting language.

A number of papers have been written on special-purpose languages built within functional languages. The idea is associated with "combinator-style programming," in which a number of higher-order functions are defined which, together with the host language, form a functional language for a new domain. Examples are the functional parsing combinators of Hutton [7]; the "geometric region server" language reported in Carlson et al. [4]; the Haskore music notating language of Hudak et al. [6]; and the picture specification language reported by Finne and Peyton Jones [5]. Yet none of them make the observation that a set of combinators very similar to those used to performed the functional computation could be defined so as to generate conventional-language code.

As a dual data point, Spinellis [12] describes the the programming of the Glasgow Haskell compiler as an exercise in program generator construction, but fails to note that, using the technique described in this paper, those mini-program generators could have been written in Haskell itself, instead of Perl and C.

2 Building program generators

It is a kind of folklore in the programming language community that

$$\text{programming language} = \lambda\text{-calculus} + \text{constants}$$

Several researchers have developed languages following this approach, although it does not seem to have undergone extensive experimental verification. We view our program generators as experiments in language design; program generation just happens to be the domain of these languages.

The key to language design, according to this paradigm, is the appropriate choice of constants. We have found that a good guide to this choice lies in the expected concrete syntax of the language (in our case, the syntax of the program specification). More specifically, we find the syntax useful in helping determine the *types* of the new constants; given the correct types, the constants themselves are often easy to write.

Consider a top-down parser generator as an example. We know we will want to write down something like:

$$\texttt{A ::=}\ \alpha\ \texttt{|}\ \beta\ \texttt{|}\ \texttt{...}\ \texttt{|}\ \gamma$$

with α, ..., γ being sequences of tokens and non-terminals. Knowledge of top-down (specifically, recursive descent) parsing tells us that this rule as a whole produces a function. Thus, the value of such a rule has type "C++ function definition." The rules on the right-hand side, taken as a whole, represent the body of that function, which might lead us to conclude that they should have type "C++ statement." However, a closer look reveals that each component of the right-hand side must have somewhere to branch to if it fails to parse; thus, it has type "function from label to C++ statement." This example is completed in the next section.

As another example, consider the case of a pretty-printer specification language. In this language, one gives rules for splitting lines, indenting, and so on, and the processor generates a program that pretty-prints an abstract syntax tree. Here is a possible rule:

```
if Cond S1 S2 ==> "if (" Cond ")" indent (newline S1)
                    newline "else" indent (newline S2)
```

This says that an if statement should be displayed with the two statements indented below the `if` and the `else`, respectively.

The collection of pretty-printing rules, as a whole, should generate a function definition. But the left-hand side (the *pattern*) and the right-hand side (the *layout*) have different types. The pattern needs to produce a predicate to test whether a given AST node is of the given type, and it also needs to produce an environment binding the names occurring in it (like `Cond` above) to C++ expressions that denote specific AST nodes. Thus,

```
type Pattern = CNode -> (CPred * Env)
```

The layout needs to use the node expression and the names occurring in the environment to produce a command (which will first check if the AST node matches and then, if so, pretty-print it). Thus,

```
type Layout = CNode -> Env -> CCommand
```

This example is discussed in detail in [10], and, in simplified form, in [9].

The paradigm given at the start of this section has proven to work quite well in the domain of program generators. We continue the parsing example in the next section, and give our major example in section 4.

3 Top-down parser generator

Top-down parsing is a well-known example of combinator-style programming in functional languages, as well as a standard example of program generation in the programming community at large. We will use it as a warm-up for the more elaborate program generator in section 4.

In the usual presentation of top-down parsing in a functional language, one starts by defining the type of "parser", and then writes combinators:

```
term: token -> Parser
  ++: Parser * Parser -> Parser
  ||: Parser * Parser -> Parser
```

Then, simply writing down a grammar, with a few syntactic decorations, yields a parser. For example, the grammar

```
A ::= aBB | B
B ::= b | cA
```

is programmed by writing[1]

```
val rec A = (term "a") ++ B ++ B || B
    and B = (term "b") || (term "c") ++ A
```

It turns out that the combinators above can be defined in such a way that combinator expressions produce C++ programs. We have already described how we arrived at the types of rules (`Parser`) and of right-hand sides of rules (`RHS`). The complete code for the top-down parser generator is given in Figure 1.[2] It assumes that the input is contained in an array, and the global integer variable `current` points to the current token. (Types `CFunction`, `Label`, `CCommand`, `Token`, and `Label` are synonyms for `string`.)

For example, a right-hand side corresponding to a terminal compares the next character to that terminal and, if there is no match, jumps to the label; if there is a match, the token pointer is incremented.

```
(* term: Token -> RHS *)
fun term (t:Token) = fn (lab:Label) =>
   %`if (tokens[current] == '^(t)')
       current++;
     else
       goto ^lab;` ;
```

Thus, each non-terminal in a grammar produces a parsing function. The productions are written slightly differently from those above:

[1]This definition does not work in ML, due to eager evaluation, but it would work in Haskell. In ML, eta-abstraction must be applied, but otherwise the definition is the same.

[2]This code uses the "anti-quotation" feature of ML of New Jersey [13]. An expression of the form `` %`...^e...` ``, where `e` is a string-valued expression, is an abbrevation for `"..."^e^"..."`, where `^` is ML's string concatenation operator.

```
type Parser = CFunction;
type RHS = Label -> CCommand;

(* term: Token -> RHS *)
fun term (t:Token) = fn (lab:Label) =>
   %`if (tokens[current] == '^(t)')
       current++;
     else goto ^lab;` ;

(* nonterm: Variable -> RHS *)
fun nonterm (v:Variable) = fn (lab:Label) =>
  %`if (!parse^v()) goto ^lab;` ;

(* ++: RHS -> RHS -> RHS *)
infix 3 ++;
fun ((rhs1:RHS) ++ (rhs2:RHS))  = fn (lab:Label) =>
  %`^(rhs1 lab)
    ^(rhs2 lab)` ;

val label_counter = ref 0;
fun genLabel () =
  let val l = makestring(!label_counter)
  in label_counter := !label_counter + 1;
     "L"^l
  end;

(* ||: RHS -> RHS -> RHS *)
infix 2 ||;
fun ((rhs1:RHS) || (rhs2:RHS))  = fn (lab:Label) =>
  let val l = genLabel()
  in %`^(rhs1 l)
       return true;
    ^l: current = pos;
       ^(rhs2 lab)`
  end;

(* ::= : Variable -> RHS -> CFunction *)
infix 1 ::= ;
fun (v:Variable) ::= (rhs:RHS) =
   let val lab = %`L1000`
   in %`int parse^v () {
        int pos = current;
        ^(rhs lab)
        return true;
    ^lab:
        current = pos;
        return false;
   }`
   end;
```

Figure 1: Top-down parser generator definitions

```
"A" ::= (term "a") ++ (nonterm "B") ++ (nonterm "B")
     || (nonterm "B") ;

"B" ::= (term "b")
     || (term "c") ++ (nonterm "A") ;
```

Each produces a C++ function. Here is the function produced for `A`:[3]

```
int parseA () {
     int pos = current;
     if (tokens[current] == 'a')
         current++;
     else
         goto L0;
     if (!parseB()) goto L0;
     if (!parseB()) goto L0;
         return true;
 L0:
     current = pos;
     if (!parseB()) goto L1000;
         return true;
 L1000:
     current = pos;
     return false;
}
```

In addition to having developed a program generator in about 40 lines of ML code, we now have a *functional* program generating language. For example, using list and ML functions like `fold`, we can produce a collection of parsing rules by mapping over a list of binary operators:

```
"Exp" ::= fold (op ||)
          (map (fn bop => (nonterm "Exp") ++ (term bop)
                                        ++ (nonterm "Exp"))
               ["+", "-", "*", "/"])
          (fn lab => %`return false;`);
```

4 MSL-in-SML

The Message Specification Language, or MSL, was designed as an approach to engineering reusable code in the domain of "message translation and validation." This is a domain of programming which has been studied by software engineers [3, 11] as a model area for the study of software re-use. The problem is to translate and validate incoming messages — bit streams — from a variety of sources in a variety of formats. The translation process involves pulling bits off the incoming stream and storing appropriate values in records; validation is simply checking that the messages have the required form. In past work, the language in which translation and validation has been performed is ADA. In this paper, we use C++.

The formats of these messages are described only in semi-formal "interface control documents;" an example is given in Figure 2 of Bell et al. [3] and reproduced here in Figure 2.

[3] As in the other examples in this paper of code produced by our program generators, we have taken the liberty of "beautifying" it for better readability, to the extent of changing indentation and line breaks.

No.	Field Name	Size	Range	Amplifying Data
1	Course	3	001–360	In degrees.
			000	No value reported.
	Field Separator	1	/	Slash.
2	Speed	4	0000–5110	In knots.
	Field Separator	1	/	Slash.
3	Altitude or	0 or 2	01–99	In thousands of feet.
	Track Confidence		HH	High Confidence.
			MM	Medium Confidence.
			LL	Low Confidence.
			NN	No Confidence.
			Blank	No altitude value reported or altitude less than 1000 feet.
	Field Separator	1	/	Slash.
4	Time			Time Group
		2	00–23	Hour
		2	00–59	Minute
	End of line	1	CR	Carriage return.

Figure 2: Reproduction of Figure 2 from [3] — Sample Interface Control Document

The approach taken by the group at OGI [3]—which is the point of departure for the current work—is to define the MSL language, in which message formats can be described *formally*, thereby allowing automatic translation to ADA. Figure 3 in [3] is reproduced here in Figure 3; it is the MSL specification for the interface control document in Figure 2. This description has two parts. The *type declarations* are an abstraction of ADA record definitions. The *action declarations* describe how the incoming bit stream should be "parsed."

As discussed above, we need to determine the *types* of the various parts of a message specification. We will consider only the action declarations, since these are the parts that generate executable code, but we will include the value ranges given in the type declarations.

It turns out that we can think of each of the parts of a message specification, and the message specification as a whole, as being of type `Message`. Given a location in the input stream, a destination address, and a statement to be executed in case of failure, a `Message` produces a statement that parses the input and places it into the destination. Thus,

```
type Message = bitsource -> recordfield -> statement -> statement;
```

Having determined this type, we can write the constants fairly easily. We give two examples.

The function `asc2int` takes an integer, giving the length of the field in the incoming message, in bytes, and a pair of integers, giving the minimum and maximum numerical values of the field, and produces a `Message`. We have assumed the existence of C++ functions `inrange`, which checks that the value of the incoming message is within a given range, and `getint`, which returns the integer value of the message.

```
fun (* asc2int : int -> (int * int) -> Message *)
    asc2int (w:int) (lo:int, hi:int)
```

```
(* Type declarations *)
type Confidence_type = [High, Medium, Low, No];

type Alt_or_TC_type = [Altitude: integer(1..99),
                       Track_confidence: Confidence_type,
                       No_value_or_Alt_less_than_1000];

type Time_type = {Hour: integer(0..23), Minute: integer(0..59)};

message_type MType = {Course: integer(0..360), Speed: integer(0..5110),
                      Alt_or_TC: Alt_or_TC_type, Time: Time_type};

(* Action declarations *)
EXRaction to_Confidence = [High: Asc 2 | "HH", Medium: Asc 2 | "MM",
                           Low: Asc 2 | "LL", No: Asc 2 | "NN"];

EXRaction to_Alt_or_TC = [Altitude: Asc2int 2,
                          Track_confidence: to_Confidence,
                          No_value_or_Alt_less_than_1000: Skip 0
                         ] @ Delim "/"; (* field separator "/" *)

EXRaction to_Time = [Hour: Asc2Int 2, Minute: Asc2Int 2
                    ] @ Delim "\n"; (* CR as field separator *)

EXRmessage_action to_MType = {Course: Asc2Int 3 @ Delim "/",
                              Speed: Asc2Int 4 @ Delim "/",
                              Alt_or_TC: to_Alt_or_TC,
                              Time: to_Time};
```

Figure 3: Reproduction of Figures 2 and 3 from [3]: An interface control document, informally and in MSL

```
            (src : bitsource) (tgt : recordfield) S =
    let val ms = makestring
    in C_if(%`inrange(^(getByte src), ^(ms w), ^(ms lo), ^(ms hi))`,
            %`^(deref tgt) = getint(^(getByte src), ^(ms w));
              ^(advanceNBytes src w)`,
            S)
    end;
```

For example, suppose `bs` is a bit source, consisting of a byte array `A` and index `bit`; the latter is an index into `A`, treating `A` as an array of bits, and is divided by 8 to get at a specific byte. Suppose also that the field `f` is the field `Hour` in the record `p`. Then, the call

```
asc2int 2 (0, 23) bs f "abort();";
```

produces the statement

```
if (inrange(A[bit / 8], 2, 0, 23)) {
  (*p).Hour = getint(A[bit / 8], 2);
  bit = bit - (bit % 8) + (8 * 2);
} else {
  abort();
}
```

As another example, the operator `seq` is used to extract a sequence of message components from the bit source, all of which must be present, and store them into the fields of a structure. Specifically, its argument is a list of `Message`s and its result is a `Message`. Since each component of the message must be present, the alternative action for each message is "error." Since the entire message can fail only if one of its components fails—and since backtracking is not permitted—the message as a whole can just ignore its alternative action.

```
fun (* seq: Message list -> Message *)
    seq (m::ml) (src : bitsource) (tgt : recordfield) S =
         %`^(m src tgt (%`error_action();`)) ^(seq ml src tgt S)`
  | seq [] src tgt S = %``;
```

< The full paper contains the complete definition of MSL-in-SML, about 100 lines of Standard ML code >

Given these definitions, our version of the MSL message specification of Figure 3 is the set of SML definitions in Figure 4. Given all the definitions in MSL-in-SML, the expression

```
print (MType (newbitsource "A" "bit") (recordptr "target") "");
```

evaluates to the string shown in Figure 5.[4]

We have attempted to show that much of the value of MSL, as defined in [3], can be obtained directly in a functional language like SML. There remains some syntactic awkwardness, but in general the language illustrated in Figure 4 (MSL-in-SML) is not very much different from the one illustrated in Figure 3 (MSL).

[4] There is no corresponding figure given in [3] with which to compare our Figure 5.

```
val to_Confidence =
   alt[asc "HH" "High", asc "MM" "Medium",
       asc "LL" "Low", asc "NN" "None"
      ];

val to_Alt_or_TC =
   alt[infield "Altitude" (asc2int 2 (1, 99)),
       infield "Track_confidence" to_Confidence,
       infield "No_Value_or_Alt_less_than_1000" (skip 0)];

val to_Time =
   seq[infield "Hour" (asc2int 2 (0, 23)), infield "Minute" (asc2int 2 (0, 59)) ];

val MType =
   seq[infield "Course" (asc2int 3 (0, 360)), delim "/",
       infield "Speed" (asc2int 4 (0, 5110)), delim "/",
       infield "Alt_or_TC" to_Alt_or_TC,
       infield "Time" to_Time
      ];
```

Figure 4: The MSL-in-SML specification corresponding to the MSL specification in Figure 3

As compared to that of [3], this approach is not only simpler — producing a similar language in about 100 lines of ML — and more reliable — the code produced by a message specification is entirely in the control of the language designer — but has the added virtue that the resulting program generator is a functional language. To give a simple example of the utility of the ML features, suppose we have a class of messages that are all identical except for the delimiters between fields: messages from one source use a slash (as in Figures 3 and 4) and those from another use a semi-colon. We can write:

```
val MType = seq[..., if (messageUses "/") then delim "/" else delim ";", ...];
```

or, more generally,

```
val MessageGen = fn delimiter => seq[..., delimiter, ...];
val MType1 = MessageGen (delim "/");
```

These capabilities come for free using our approach.

5 Conclusions

We have presented a simple and elegant method of developing powerful program generators, by defining program-generating functions in a functional language. With this approach, a language can be developed with relatively modest effort. Moreover, the result is a "higher-order" language; this means the language will scale up well, allowing for well-structured and maintainable program-generation specifications, rather than being extended with a set of *ad hoc* data types and control structures, as so often happens to special-purpose languages.

```
if (inrange(A[bit / 8], 3, 0, 360)) {
  (*target).Course = getint(A[bit / 8], 3);
  bit = bit - (bit % 8) + (8 * 3);
} else { error_action(); }
if (A[bit / 8] == '/')
  bit = bit - (bit % 8) + 8;
else error_action();
if (inrange(A[bit / 8], 4, 0, 5110)) {
  (*target).Speed = getint(A[bit / 8], 4);
  bit = bit - (bit % 8) + (8 * 4);
} else { error_action(); }
if (A[bit / 8] == '/')
  bit = bit - (bit % 8) + 8;
else error_action();
if (inrange(A[bit / 8], 2, 1, 99)) {
  (*target).Alt_or_TC.Altitude = getint(A[bit / 8], 2);
  bit = bit - (bit % 8) + (8 * 2);
} else {
  if ((A[bit / 8] == 'H') && (A[bit / 8 + 1] == 'H')) {
    (*target).Alt_or_TC.Track_confidence = High;
  } else {
    if ((A[bit / 8] == 'M') && (A[bit / 8 + 1] == 'M')) {
      (*target).Alt_or_TC.Track_confidence = Medium;
    } else {
      if ((A[bit / 8] == 'L') && (A[bit / 8 + 1] == 'L')) {
        (*target).Alt_or_TC.Track_confidence = Low;
      } else {
        if ((A[bit / 8] == 'N') && (A[bit / 8 + 1] == 'N')) {
          (*target).Alt_or_TC.Track_confidence = None;
        } else {
          (*target).Alt_or_TC.No_Value_or_Alt_less_than_1000 = 1;
        }
      }
    }
  }
}
if (inrange(A[bit / 8], 2, 0, 23)) {
  (*target).Time.Hour = getint(A[bit / 8], 2);
  bit = bit - (bit % 8) + (8 * 2);
} else { error_action(); }
if (inrange(A[bit / 8], 2, 0, 59)) {
  (*target).Time.Minute = getint(A[bit / 8], 2);
  bit = bit - (bit % 8) + (8 * 2);
} else { error_action(); }
```

Figure 5: Code produced by evaluating MSL-in-SML specification in Figure 4

However, the method is not without its drawbacks, some apparent and some subtle, which are the subject of current research. One obvious problem is that the languages are not as clean syntactically as they would be if developed in a more conventional way; for example, a conventional parser generator does not require that you put the word `term` before every occurrence of a terminal symbol. Some of these cases can be addressed through a simple lexical preprocessing step, but some cannot.

Another shortcoming of the method is seen in the top-down parsing example. There, the natural recursion that was used in the functional version of the parser had to be replaced by a separate operator that produced a C++ function definition (which was, in turn, recursive). It would be more pleasant if the host functional language were able to compute fixed-points in the chosen domain, or could be modified to do so; here, we are talking about the domain of programs, where the fixed-point operation is the one that creates a recursive function in the object code.

A still deeper problem is apparent in the pretty-printing example, discussed in section 2. Suppose that in the pretty-printing rule for `if` we want to decide whether to skip a line after the `else` based on the size of the following statement. Assume we have a function `size: Name -> CNode -> Env -> int`. We are tempted to write:

```
if Cond S1 S2 ==> ... (if ((size S2) > 10) then newline else "") ...
```

where the `if` on the right is ML's `if`. However, this does not work. `(size S2)` is of type `CNode -> Env -> int`, so the comparison `(size S2) > 10` is a type error. In effect, the `>` operator — as well as the constant `10`, the `if`, and, apparently, the entire ML language — needs to be lifted to type `CNode -> Env ->` *whatever*. This is an example of the well known concept of a monad [14], but the problem is that we have no way to lift the entire ML language to this monad, nor to switch from ML's normal operation to this monad.

Perhaps the deepest problem here is that we have no way to determine the adequacy of the chosen set of operators. If the set of operators is not powerful enough, users will be unable to use the language to solve all of their problems; either that, or they will have to deal with the underlying representation of the new values. At present, there seems to be no way even to formalize the notion of adequacy.

These problems that we have encountered in building program generators — producing convenient syntax, calculating fixed-points, incorporating new monads, and finding an adequate set of operators — are by no means specific to the domain of program generators. We have seen them arise in other domain-specific languages as well. In this sense, program generation is simply a domain, familiar to programmers and students of programming languages, that contains a ready supply of special-purpose languages. These languages can be used to study the problems of language design, and more specifically, to test the thesis that languages can be designed by adding constants to the λ-calculus.

References

[1] S. Ahmed, D. Gelernter, *Program builders as alternatives to high-level languages,* Yale Univ. C.S. Dept. TR 887, November 1991.

[2] B. Balzer, N. Goldman, D. Wile, *Rationale and Support for Domain Specific Languages*, USC/Information Sciences Institute, available at `http://www.isi.edu/software-sciences/dssa/dssls/dssls.html`.

[3] J. Bell, F. Bellegarde, J. Hook, R.B. Kieburtz, A. Kotov, J. Lewis, L. McKinney, D.P. Oliva, T. Shear, L. Tong, L. Walton, and T. Zhou, *Software design for reliabiity and reuse: A proof-of-concept demonstration*, TRI-Ada '94.

[4] W. E. Carlson, P. Hudak, M. P. Jones, *An experiment using Haskell to prototype "Geometric Region Servers" for navy command and control*, Research Report YALEU/DCS/RR–1031, Yale Univ. C. S. Dept., May 1994.

[5] S. Finne, S. Peyton Jones, *Pictures: A simple structured graphics model*, Dept. of Computing Science, Univ. of Glasgow, 1996.

[6] P. Hudak, T. Makucevich, S. Gadde, B. Whong, *Haskore music notation: An algebra of music*, J. Func. Prog., to appear.

[7] G. Hutton, *Higher-order functions for parsing*, J. Func. Prog. 2(3), 323–343, July 1992.

[8] S. Kamin, *Report of a workshop on future directions in programming languages and compilers*, ACM SIGPLAN Notices 30 (7), July 1995, 9–28.

[9] S. Kamin, *The Challenge of Language Technology Transfer (Position Paper)*, Computing Surveys 28A(4), 1996.

[10] S. Kamin, *ML as a Meta-Programming Language*, Univ. of Illinois, Sept. 1996, available at `www-sal.cs.uiuc.edu/~kamin/pubs`.

[11] C. Plinta, K. Lee, and M. Rissman, *A model solution for C^3I message translation and validation*, Technical Report CMU/SEI–89–TR–12, Software Engineering Institute, Carnegie Mellon University, December 1989.

[12] D. Spinellis, *Implementing Haskell: Language implementation as a tool building exercise*, **Software: Concepts and Tools** 14, 1993, 37–48.

[13] . *Standard ML of New Jersey User's Guide*, February 1993.

[14] P. Wadler. *The essense of functional programming*, Proc. 19th ACM Symp. on Principles of Programming Languages, Albuquerque, NM, Jan. 1992, 1–14.

[15] L. Walton and J. Hook, *Message Specification Language (MSL): Reference Manual, Revision: 1.8*, Oregon Graduate Institute, Oct. 6, 1994.

[16] R. C. Waters, *The Programmer's Apprentic: A session with KBEmacs*, IEEE Trans. Software Eng. SE–11(11), 1296–1320, Nov. 1985.