No Assembly Required: Compiling Standard ML to C

David Tarditi Anurag Acharya Peter Lee

November 1990
CMU-CS-90-187

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

C has been proposed as a portable target language for implementing higher-order languages. Previous efforts at compiling such languages to C have produced efficient code, but have had to compromise on either the portability or the preservation of the tail-recursive properties of the languages. We assert that neither of these compromises is necessary for the generation of efficient code. We offer a Standard ML to C compiler, which does not make either of these compromises, as an existence proof. The generated code achieves an execution speed that is just a factor of two slower than the best native code compiler. In this paper, we describe the design, implementation and the performance of this compiler.

This research was partially supported by the National Science Foundation under PYI grant #CCR-9057567. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation or the US Government.

Keywords: Standard ML, Continuation-passing style, compiling functional languages, retargetable code, compilation to C

1 Introduction

Implementors of new programming languages are faced with a dilemma: whether to sacrifice efficiency for portability or the other way around. One approach that has been successfully used in the past to avoid writing a code generator for each architecture is to compile to C, using C as a universal intermediate language. C makes an efficient intermediate language because it is relatively close to assembly language, yet mostly machine independent, and compilers for C are available on most machines.

Bartlett [7] has shown that it is possible to compile Scheme programs into efficient C programs. In his approach, constructs in Scheme are mapped to apparently similar constructs in C. Unfortunately, this conceptually simple approach leads to several compromises. First, the Scheme implementation fails to reflect the pragmatics of the language: proper tail-recursion is lost, even though a compile-time analysis is used to recover some tail-recursion information. Second, features such as first-class continuations (call/cc) and garbage collection require the use of assembly language. The whole point of compiling to C, though, is to avoid such machine dependencies.

Thus, Bartlett's implementation raises several questions. How can Scheme-like languages be compiled without any use of assembly language? Can tail-recursive languages such as Scheme be compiled to C and retain the property of proper tail-recursion? Can this be done without an unacceptable loss of efficiency?

In this paper, we answer these questions by describing our experience with compiling Standard ML [12], a language which has a dynamic semantics similar to that of Scheme, to C. We have been able to successfully compile all of Standard ML into efficient and portable C code which runs on 32-bit architectures[1]. In doing so, we have completely avoided the use of assembly language. In addition, our code generator handles extensions to Standard ML for call/cc and asynchronous signal handling. Our code is typically about two times slower than the code produced by the best available Standard ML compiler, which compiles to native machine code. Our implementation runs on Sun-3s, Sparcstations, Decstations and a 80486-based machine. To our best knowledge, no full implementation of Standard ML is available for the 80x86-based machines.

We discuss the basic approach we used for compiling ML to C and our reasons for choosing this approach. We then discuss some limitations of the approach and the optimizations we used to significantly improve the performance of the generated C code. We present a series of benchmarks to support our claim of having achieved good performance relative to the best available native code compiler. Finally, we present our conclusions.

2 Background

2.1 Related Work

Besides Bartlett's work, other related work includes the portable Cedar project [6]. In this work, an existing compiler for Cedar was successfully retargeted to C in order to port existing code to new systems. They report achieving very good code performance for their implementations on the SPARC and Motorola 68020.

[1]The runtime system, however, currently requires some version of the Unix 4.3 BSD operating system.

Our work differs significantly from this work in several aspects. Cedar does not have the requirement of proper tail-recursion and does not have functions as first-class values. Their implementation used highly machine and C compiler dependent code to traverse the C call stack to implement exception handling. Our implementation does not use any machine-dependent code and is completely portable across a wide class of machines.

2.2 Standard ML

Standard ML (SML) is a modern programming language embodying many innovations in programming language design. It is a lexically-scoped mostly-functional programming language; functions are first-class values. It is statically-typed but unlike other statically-typed languages (Pascal, Ada), the types are automatically inferred by the compiler and the type system is polymorphic. It has a sophisticated modules mechanism for developing large programs which typechecks the interface between modules, like in Ada or Modula. It provides a type-safe dynamically-scoped exception mechanism to handle unusual or deviant conditions. It provides garbage collection, like most other languages of its type. It also provides complete runtime safety — in particular, programs never "dump core".

As the can be seen from the previous paragraph, there is considerable semantic distance between Standard ML and C. Features like dynamically-scoped exceptions, garbage collection, complete run-time safety, and higher-order functions all pose problems for a C implementation of SML.

3 Compilation Strategy

Languages like Standard ML and Scheme can be regarded as syntactically sugared versions of the λ-calculus. The general strategy for compiling such languages is to unsugar programs to a simple call-by-value λ-calculus augmented with a branching operation, record operations, and a set of primitive operators. In the case of Standard ML, the conversion also removes the type information. This reduces the problem of compiling Standard ML to a more manageable size, since this λ-calculus is much smaller.

There are two basic approaches to compiling the λ-calculus. The first approach is to use *continuation-passing style* (CPS). This approach has been used successfully in Rabbit, Orbit and the Standard ML of New Jersey compiler [15, 11, 3]. CPS uses a λ-calculus based intermediate representation meeting the invariants that function applications never be nested and that function calls always be tail-recursive. Since function calls are always tail-recursive, the first call to return is also the only call to return. Thus, a function call is transformed into a goto with arguments. Any program in the λ-calculus language can be converted into CPS by a simple $O(n)$ transformation[9].

There are several advantages of using CPS as an intermediate representation [15, 10]. First, all intermediate values are explicitly named. Second, control-flow is explicitly available via continuations. This makes it easy to implement exceptions and call/cc. Third, since all tail-calls turn into jumps, tail-recursion elimination is achieved for free. Fourth, the target machine for a CPS program is simple. It requires a set of registers, a heap on which one can allocate records, a jump instruction, and a small set of instructions which can be used to implement the CPS primitive operators [3].

The second approach to compiling the λ-calculus is to use a stack to hold activation records for functions as the functions are evaluated. When the evaluation of an expression yields a function, a closure is built. When this function is applied, the values for the lexically scoped variables are fetched from the enviornment part of this closure.

The stack-based approach is more complicated. Recall that the exception mechanism of ML is dynamically-scoped. Therefore when an exception occurs the stack must be unwound until an exception handler for the particular exception is found. Using a stack also makes it difficult to preserve proper tail recursion. If the last thing a function f does is call another function g, then the new activation record for g must replace the activation record for f on the stack. Finally, the presence of a stack complicates the implementation of the call/cc extension to Standard ML [8].

These complications become serious problems if we try to map ML constructs to apparently similar C constructs. We can map ML functions to corresponding C functions by flattening out all functions to a single lexical level. Closures can be represented by using a record containing a C function pointer and the environment for the function[2]. This, however, does not solve any of the problems mentioned above. First, one must implement some cumbersome method for handling exceptions. Second, one has lost the property of proper tail recursion. It is trivial to fix this for functions which are immediately tail recursive (that is, they call themselves). It is not obvious how to fix this for mutually recursive tail-calls and tail-calls to functions that are not known until runtime, which are both quite frequent in a programming style using higher-order functions. Finally, it is impossible to implement the call/cc extension portably in C, since there is no portable way to save the stack of the entire program, and then restore the stack at an arbitrary point in time. [3]

4 Design of the Compiler

Based on the analysis presented in the previous section, we chose to use the CPS approach to code generation and to treat C as a target assembly language.

Appel and Jim [3] proposed a variant of CPS that they called continuation-passing, closure-passing style. This is a refinement of the approach to code generation taken in Orbit [10]. In this variant, all functions have been flattened out to one lexical level and record operations are used to explicitly represent closures.

An example of the transformation is shown in Figures 1 and 2. Figure 1 contains SML code and Figure 2 contains the representation of the same code after the transformation.

The program in Figure 2 is presented in a pseudo-code style: we have not expanded infix operators to their true CPS form. Note that the converted code looks remarkably similar to C code, except for the fact that all function calls are tail-recursive.

The continuation-passing, closure-passing style approach to code generation has been implemented in the Standard ML of New Jersey (SML/NJ) compiler [4], a publicly available, freely redistributable optimizing compiler developed at AT&T and Princeton University. It is currently the premier optimizing compiler for Standard ML. This compiler is organized into a series of modules which repeatedly transform the program in various intermediate

[2] Bartlett used a similar approach in his Scheme—C compiler.

[3] Longjmp and setjmp cannot do this!

```
fun foo x =
   let fun bar 0 = "bar"
         | bar x = baz(x-1)
       and baz 0 = "baz"
         | baz x = bar(x-1)
   in bar x
   end
```

Figure 1: SML code before transformation

```
fun fool(x,c) = barl(x,c)
and barl(x,c) = if x=0 then c "bar" else bazl(x-1,c)
and bazl(x,c) = if x=0 then c "baz" else barl(x-1,c)
```

Figure 2: SML code after transformation

languages, the end result being functions represented in the continuation-passing, closure-passing style. This makes it easy to replace modules at different stages in the compilation.

We chose to reuse the SML/NJ modules that precede the code-generation stage, namely the front-end, CPS conversion and CPS optimization phases. This allowed us to focus our attention on the issues involved in translation to C. It also allowed us to directly compare the performance of our implementation with the best available native code generator, factoring out any differences that might arise due to different compilation strategies. In addition, this permitted us to reuse the well-designed runtime system of SML/NJ [2].

This decision reduced our problem to compiling the CPS representation of SML programs as generated by the SML/NJ compiler. The CPS representation is shown in Figure 3.

Continuation expressions, or values of type `cexp`, represent programs. Most variants of the `cexp` type bind zero or more variables whose scope is another continuation expression. Variable names are represented by values of type `lvar`. In practice, each variable name is designated by a unique integer.[4]

The target machine required to efficiently implement this language is very simple. The machine configuration consists of:

- A heap, *i.e.*, a contiguous block of memory
- A set of general purpose registers
- Reserved registers for holding the following:
  - Heap pointer
  - Heap limit pointer
  - Current exception continuation
  - Arithmetic temporaries

[4]For more information about the CPS language, refer to [3].

```
datatype value = VAR of lvar
       | LABEL of lvar
       | INT of int
       | REAL of string
       | STRING of string

datatype cexp = RECORD of (value * accesspath) list * lvar * cexp
      | SELECT of int * value * lvar * cexp
      | OFFSET of int * value * lvar * cexp
      | APP of value * value list
      | FIX of (lvar * lvar list * cexp) list * cexp
      | SWITCH of value * cexp list
      | PRIMOP of primop * value list * lvar list * cexp list
```

Figure 3: CPS language used by Standard ML of New Jersey

The instruction set of the target machine is very close to those found on most conventional machines. It includes load, store, logical operations, arithmetic operations with and without overflow checking, floating operations, register-register move, conditional branches and jumps. The instructions can be labeled.

5 Implementation

It is straightforward to implement the target machine resources in C. Registers are implemented using global variables and the heap is implemented by an array of integers.

Most target machine instructions are also straightforward to implement in C. The only ones that presented any problems were the jump instruction [5] and arithmetic operations with overflow checking. The implementation of the jump instruction is problematic because labels are not first-class values in C. In particular, we cannot store a label in memory or jump to it from an arbitrary point in the C program. The only way to get the address of a block of C code is to encapsulate it in a C function. Since jumps are not supposed to return, we cannot use C function calls to implement jumps; for if we did, the stack would quickly overflow.

Instead, we use a technique from Rabbit which Steele called the *UUO handler* and which we refer to as the *apply-like* procedure. The apply-like procedure emulates the apply operation. Code execution begins with the apply-like procedure calling the first function to be executed. When this function wants to call another function, it returns to the apply-like procedure with the address of the next function to call. In general, when a function f needs to call another function g, it returns the address of g to the apply-like procedure, which then calls g. In this way, the depth of the stack never grows to more than 2.

The implementation of arithmetic instructions with overflow checking is difficult since implementations of C tend to ignore overflow. For Standard ML, however, overflow

[5] Recall that function calls are tail-recursive after CPS conversion and are equivalent to jumps with arguments passed in registers.

```
int fool(),bar1(),baz1();

int apply(start)
int (*start)();
{ while (1)  start = (int (*)()) (*start)();}

int fool ()
{ return((int) bar1); }

int bar1 ()
{ if (R1==0) { R1 = "bar"; return R2; }
  else { R1 = R1 - 1; return ((int) baz1); }
}

int baz1 ()
{ if (R1==0) { R1 = "baz"; return R2); }
  else { R1 = R1 - 1; return ((int) bar1); }
}
```

Figure 4: C code and apply-like procedure

checking is mandated by the language definition. Given the usual ranges for integers on two's complement machines, addition, subtraction, multiplication, and division can all overflow. To implement overflow checking, we had to add explicit checks which used bitwise operations on the operands and the result.

For register allocation, we reused the algorithm used in SML/NJ. The algorithm is divided into the *spilling* phase and the *register assignment* phase. In the spilling phase, the CPS programs are rewritten before instruction selection so that no subexpression has more than n free variables, where n is related to the number of actual registers. [6] This guarantees that every variable, at any point in the program, can be placed in a register. Register assignment is then done on the fly using register-tracking. To assign a register to a variable, we calculate the set of registers that are already used by variables that are free in the body of the CPS expression binding the variable. We then choose some unused register for the variable. The register assignment phase uses a variety of heuristics to try to minimize shuffling of registers at function calls.

Figure 5 contains a simplified version of the C code that would be generated for the sample SML code shown in Figure 1. The figure also shows the apply-like procedure stripped of its initialization code.

6 Benchmarks

We benchmarked three versions of our system and the SML/NJ compiler on two different classes of architectures. The three versions are unoptimized (Unopt), optimized (Opt-s) and

[6] In our case, we set the number of registers to 31. This number arises from the runtime system

optimized with unsafe arithmetic (Opt-u). The unoptimized version is a straightforward implementation of the design described in Section 5. The optimized version uses the techniques presented in Section 7 to optimize the generated C code. The optimized version with unsafe arithmetic gives an idea of the performance penalty imposed by the unavailability of arithmetic operations with overflow in C.

The benchmark suite is a combination of real programs in regular use and programs created to exercise particular aspects of implementations. The suite is:

- **insert:** insertion sort of a pseudo-randomly generated list of 2000 integers.
- **fib:** typical doubly exponential version of fibonacci used in introductory programming classes.
- **ml-yacc:** Parser generator based on the Unix *yacc* utility running on the actual parser specification for SML/NJ [16]. It is approximately 6500 lines of SML code.
- **ml-lex:** Lexer generator based on the Unix *lex* utility running on the actual lexer specification for SML/NJ [5]. It is approximately 1200 lines of SML code.

ml-lex and *ml-yacc* are part of the SML/NJ distribution and are in widespread use. For example, they are used to generate the parser and the lexer for SML/NJ itself. We plan to add several more benchmarks in the near future. They include the sml2c compiler compiling a medium-sized ML program, a benchmark to exercise call/cc and integer matrix multiplication.

We ran the benchmarks on a Sun 3/140 with 16 Mbytes of memory and and a Decstation 5000 with 32 Mbytes of memory. We used built-in timing functions available in SML/NJ which are based on the Unix *getrusage* utility. We factored out the garbage collection costs, so our numbers represent only code speed. Garbage collection costs are irrelevant for the sake of comparison since all versions of code generated by sml2c and SML/NJ generate the same amount of garbage and use the same garbage collector.

The C code was compiled on the Sun-3 using *gcc* [14] with the *-O* flag to enable optimization. On the Decstation 5000, we used the MIPS C compiler with the *-O* option. The benchmark results are tabulated in Tables 1 and 2.

We found that our benchmark programs were about twice as slow as the native code programs produced by SML/NJ. The running times of the real programs with safe arithmetic were not significantly worse than those with unsafe arithmetic. This indicates, of course, that the programs spend most of their time doing symbolic computation and very little time doing arithmetic.

We also compared the size of our stand-alone executables with those generated by SML/NJ for both the Sun-3 and the Decstation 5000. The executables were stripped of symbol table information to obtain a better measure of the code size. Sizes of the executables for each of the versions benchmarked can be found in Tables 3 and 4. We found that the size of our executables with optimizations and safe arithmetic was well within a factor of two of those produced by SML/NJ.

Benchmarks	SML/NJ (α)	Opt-u (β)	β/α	Opt-s (δ)	δ/α	Unopt (γ)	γ/α
insert	10.8s	20.8s	1.92	19.9s	1.84	37.5s	3.47
fib	20.0s	40.0s	2.00	50.6s	2.53	110.0s	5.50
ml-yacc	45.6s	88.2s	1.93	89.2s	1.95	141.8s	3.10
ml-lex	163.4s	276.8s	1.69	273.7s	1.67	464.7s	2.83

Table 1: Benchmarks for Sun 3/140

Benchmarks	SML/NJ (α)	Opt-u (β)	β/α	Opt-s (δ)	δ/α	Unopt (γ)	γ/α
insert	2.14s	3.39s	1.58	3.36s	1.57	5.80s	2.71
fib	2.73s	6.82s	2.50	7.94s	2.91	11.15s	4.08
ml-yacc	4.94s	9.20s	1.86	9.80s	1.98	16.85s	3.41
ml-lex	15.43s	26.95s	1.74	26.81s	1.74	50.36s	3.26

Table 2: Benchmarks for Decstation 5000

Benchmarks	SML/NJ (α)	Opt-u (β)	β/α	Opt-s (δ)	δ/α	Unopt (γ)	γ/α
insert	197K	279K	1.41	279K	1.41	418K	2.12
fib	172K	279K	1.62	279K	1.62	418K	2.43
ml-yacc	409K	631K	1.52	631K	1.52	983K	2.40
ml-lex	229K	377K	1.64	385K	1.68	582K	2.51

Table 3: Size of executables for Sun 3/140

Benchmarks	SML/NJ (α)	Opt-u (β)	β/α	Opt-s (δ)	δ/α	Unopt (γ)	γ/α
insert	295K	332K	1.12	335K	1.13	507K	1.72
fib	274K	332K	1.21	335K	1.22	507K	1.85
ml-yacc	610K	844K	1.38	856K	1.40	1348K	2.21
ml-lex	360K	479K	1.33	491K	1.36	745K	2.07

Table 4: Size of executables for Decstation 5000

7 Optimizations

There are a number of problems with the implementation described in the Section 5. The generated C code fails to make effective use of registers, functions calls are expensive, and arithmetic is expensive.

Recall that the target machine registers are implemented using global variables. Most C compilers will not move global variables into real registers[13]. This is true for a number of reasons. First, they cannot be sure whether any write via a pointer can possibly affect global variables since they compile only on a file-by-file basis. Some other separately compiled file may capture the address of a global variable and then pass it on to code in another file. Second, moving global variables into registers affects the semantics of signals. If a global variable is moved to a register for the duration of the execution of a function, it cannot be updated by any signal handler for the execution of a function. Since global variables are presumably the only way signal handlers can communicate to normally executing code, this is problematic.

Functions calls are expensive since every function call involves a return to the apply-like procedure and then a call by the apply-like procedure. This involves two jumps. The expense is usually more than this, since many C compilers implement the return by a jump to a piece of code at the end of a function which cleans things up and then does the actual jump back to the apply-like procedure. In addition, if we did have values in registers, we might incur some expense to save register values when entering a function and to restore register values when exiting the function. This actually occurs in practice on callee-save systems even though our apply-like procedure has no variables live across function calls. Thus, the true cost of a function call is usually three jumps and some indeterminate number of loads and stores.

Finally, integer arithmetic is expensive because most C compilers provide no way to use the integer overflow signal. Thus, we incur an expense of doing explicit software overflow checks. Addition and subtraction require complicated explicit bitwise tests and a function call is needed for multiplication. For division, some comparisons and branches suffice.

We designed and implemented a series of optimizations to deal with these problems. We present them in the following subsections.

7.1 Register Caching

To make effective use of real registers, we cache target machine registers in local variables for the duration of function calls. Most C compilers attempt to place local variables in registers when possible. This caching is done when it is worthwhile.

Our register caching optimization uses a simple static count of the number of uses of a register in a function body to decide whether to cache a target machine register for the duration of the function call. If the number of uses is greater than 2, a small arbitrarily chosen integer, we cache the register. Caching of certain registers, notably the heap pointer register and exception continuation register, that must be valid whenever machine fault might occur may cause a problem. We handle this by never caching the exception continuation register and spilling the heap pointer register to the corresponding global variable before executing any instruction that may cause a machine fault. Fortunately, the only instructions

which may cause machine faults in our implementation are division by zero and floating point operations, so the need to spill is fairly rare.

The register caching optimization could be improved, especially for small functions, by doing some flow analysis to detect loops. In addition, we should only count uses along each possible path through a function body, not the entire function body. We could also do some peephole optimization to eliminate situations where we first write to a local variable, and then write the local variable to a global variable. C compilers will typically not optimize this, for the same reasons that they will not move global variables into registers.

7.2 Function integration

A *known* function is a function whose call sites are all known. If all call paths to a known function f pass through a single function g, we can integrate f into the body of g. This means that f gets placed in the body of g and that all calls to f turn into gotos. We also perform direct tail-recursion elimination, turning all calls to g within its body to gotos. Function integration is useful for avoiding passes through the *apply-like* function. In particular, it is useful for optimizing tight tail-recursive loops.

Computing whether all call paths to a known function pass through another function can be cast as the classical problem of computing dominators in a call graph with multiple start nodes, where each unknown function is a start node. An algorithm for computing dominator information can be found in [1]. A maximal dominator is defined to be a function which dominates itself. This captures the intuitive concept of the highest-level dominator. After computing the set of dominators for each known function, we can integrate the known function into its maximal dominator provided that its maximal dominator is not itself.

Consider the SML code in Figure 1. The functions `bar` and `baz` are known and `foo` is the maximal dominator for them. Figure 2 shows the pseudo-CPS code that represents these functions as they are presented to the code generator. Figure 5 shows a simplified version of C code generated for these functions with function integration and register caching. Note that the mutually tail-recursive functions have been compiled into a tight loop with the loop counter placed in a register. [7]

7.3 Modified overflow checks

We can simplify the overflow checks by constant-folding them in the case where one operand is known at compile time. For example, the overflow check for addition can be reduced to a single comparison and branch from a complicated boolean expression involving 5 operators and the procedure call for multiplication can be replaced with two compares and a branch.

7.4 Benchmarks

To show the benefits of the optimizations, we compiled and timed our benchmark programs, enabling only one optimization at a time. We measured the performance of the register caching (reg), function integration (integ), and arithmetic optimizations (arith). We also measured the performance of the tail-recursion elimination optimization without function

[7] The redundant gotos appearing in the code are optimized away by most C compilers.

```
int fool();
int fool ()
{ register int r1;
  r1=R1;
 goto bar1;
bar1:
  if (r1==0) { r1 = "bar"; R1=r1; return ((int) R2); }
  else { r1 = r1 - 1; goto baz1; }
baz1:
  if (r1==0) { r1 = "baz"; R1=r1; return ((int) R2); }
  else { r1 = r1 - 1; goto bar1; }
}
```

Figure 5: Optimized C code

Benchmarks	unopt	reg	integ	tail	arith	opt
insert	37.5s	28.6s	32.3s	30.1s	34.7s	19.9s
fib	110.0s	76.7s	92.6s	101.3s	86.3s	50.6s
ml-yacc	141.8s	118.5s	131.4s	138.2s	134.8s	89.2s
ml-lex	464.7s	419.2s	432.5s	462.0s	469.7s	273.7s

Table 5: Benchmarks for optimizations (Sun 3/140)

integration (tail), to see whether function integration is an improvement over tail-recursion elimination. For the purposes of comparison, we compiled the benchmark programs with no optimizations (unopt) and with all optimizations and safe arithmetic (opt). The benchmark results are shown in Table 5. We ran the experiments on a Sun-3/140. The C code was compiled by *gcc* with the *-O* option.

To demonstrate whether our optimizations enhance or interfere with each other, we have listed the sum of the speed-ups over unoptimized code for each individual optimization versus the speed-up for all optimizations over unoptimized code in Table 6.

We see that in all cases, except *fibonacci*, the speedup obtained with all optimizations is greater than the sum of the speedups for individual optimizations. This is to be expected from the interaction between function integration and register caching. With function integration, we need to spill values cached in local variables to global variables less frequently, especially for tight loops. We attribute the dramatic speed-up in the lexer generator to this interaction. We speculate that the interference shown in *fibonacci* is due to the poor performance of the

Benchmarks	$\sum$ individual speed-ups	speed-up (all optimizations)
insert	16.9s	17.6s
fib	74.4s	59.4s
ml-yacc	40.7s	52.7s
ml-lex	72.7s	191.0s

Table 6: Sum of individual speed-ups versus speed-up for all optimizations at once

static textual count heuristic (for register caching) when the function integration optimization is enabled.

8 Conclusions and Future Work

We can now answer the questions that were raised at the beginning of this paper. Yes, it is possible to compile Scheme-like languages without using any assembly language. And, yes, it is possible to develop portable implementations of such languages without sacrificing either proper tail-recursion or efficiency. We offer our system, *sml2c*, as an existence proof.

We would like to point out that even though C is quite close to assembly language, there is sufficient semantic distance between the two to create significant obstacles to using C as a target language. The biggest problems are the lack of first-class labels and the arbitrary limits placed on input programs by most C compilers. [8]

We have several potential optimizations that we plan to evaluate and possibly incorporate into the compiler. We also plan to build a tool to generate a detailed trace of program execution at CPS level. We hope to use this tool to perform quantitative analyses of proposed and implemented optimizations.

9 Acknowledgements

We would like to thank Puneet Kumar for helping us obtain some of the benchmark numbers.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison–Wesley Publishing Company, 1986.

[2] A. W. Appel. A Runtime System. *Lisp and Symbolic Computation*, 3(4):343–380, Nov. 1990.

[3] A. W. Appel and T. Y. Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th Annual ACM Symbol on Principles of Programming Languages*, pages 293–302. ACM, Jan. 1989.

[4] A. W. Appel and D. B. MacQueen. A Standard ML Compiler. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274, pages 301–324. Springer-Verlag, 1987.

[5] A. W. Appel, J. Mattson, and D. Tarditi. A Lexical Analyzer Generator for Standard ML. Unpublished. This manual is distributed with the Standard ML of New Jersey compiler.

[8] For those who are interested, we plan to describe the C compiler problems we encountered in a future technical report.

[6] R. Atkinson, A. Demers, C. Hauser, C. Jacobi, P. Kessler, and M. Weiser. Experience creating a portable cedar. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 322–329. ACM, 1989.

[7] J. F. Bartlett. SCHEME→C: A Portable Scheme-to-C Compiler. Technical report, DEC Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, CA 94301 USA, Jan. 1989.

[8] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77. ACM, 1990.

[9] R. A. Kelsey. *Compilation By Program Transformation*. PhD thesis, Yale University, May 1989.

[10] D. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Yale University, Feb. 1988.

[11] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: An optimizing compiler for scheme. In *Proceedings of the SIGPLAN '86 Conference on Programming Language Design and Implementation*, pages 219–233. ACM, 1986.

[12] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990. This is a formal definition of Standard ML.

[13] V. Santhanam and D. Odnert. Register allocation across procedure and module boundaries. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 28–39. ACM, June 1990.

[14] R. M. Stallman. Using and Porting GNU CC. GNU CC is a widely available C compiler developed by the Free Software Foundation, Sept. 1989.

[15] G. L. Steele Jr. Rabbit: A compiler for Scheme (a study in compiler optimization). Master's thesis, AI Laboratory Technical Report AI-TR-474, Massachusetts Institute of Technology, May 1978.

[16] D. Tarditi and A. W. Appel. ML-Yacc, version 2.0: Documentation for Release Version. Unpublished. This manual is distributed with the Standard ML of New Jersey compiler.