Pre-Scheme: A Scheme Dialect for Systems Programming

Richard A. Kelsey
NEC Research Institute
kelsey@research.nj.nec.com

June 4, 1997

Abstract

Pre-Scheme is a statically typed dialect of Scheme that gives the programmer the efficiency and low-level machine access of C while retaining many of the desirable features of Scheme. The Pre-Scheme compiler makes use of type inference, partial evaluation and Scheme and Lisp compiler technology to compile the problematic features of Scheme, such as closures, into C code without significant run-time overhead. Use of such features in Pre-Scheme programs is restricted to those cases that can be compiled into efficient code. Type reconstruction is done using a modified Hindley/Milner algorithm that allows overloaded user-defined functions. All top-level forms in Pre-Scheme programs are evaluated at compile time, which gives the user additional control over the compiler's partial evaluation of a program. Pre-Scheme has been implemented and used to write a byte-code interpeter and associated support code for a complete Scheme implementation.

1 Introduction

High-level programming languages, such as Scheme [15], Haskell [6], and ML [11], don't work well for writing programs that require maximum performance or direct access to machine-level instructions and data structures. This is deliberate in some circumstances, such as checking at run-time that array references are in range, and unavoidable in others, for example the run-time overhead required for garbage collection. Pre-Scheme is a statically typed dialect of Scheme that avoids all such overhead while attempting to preserve the features of Scheme as much as possible.

The most important Scheme feature that is preserved is Scheme's semantics. Pre-Scheme's semantics are identical to Scheme's, with the caveat that a Pre-Scheme program may run out of space, because it lacks both a garbage collector and full proper tail-recursion. A Pre-Scheme program will produce the same answer when compiled as when run in a Scheme implementation, if the compiled program does not run out of space.

The programs for which Pre-Scheme is appropriate are usually written in a low-level language, for example C or Pascal. Doing so gives the desired access to the machine, but only by giving up all of the advantages of using a higher-level language. Pre-Scheme is an attempt to get the best of both worlds by restricting Scheme to those programs that when compiled run with no more overhead than equivalent programs written in low-level languages. Obviously we want the minimum necessary restrictions so as to preserve as much of Scheme's expressiveness as possible. Pre-Scheme provides the following features lacking in C:

- local proper tail recursion
- nontrivial syntactic extensions (macros)
- higher order procedures
- type-checked polymorphism
- interactive debugging
- a module system
- Scheme semantics

Local proper tail recursion is necessary, since recursion is the only way to express iterative constructs in Scheme or Pre-Scheme. Pre-Scheme inherits Scheme's macro facility. C has its own macros, but they are textual as opposed to syntactic, and are much less powerful and convenient than Scheme macros. Proper tail recursion is implemented so long as the compiler can translate it into an iterative C construct, or when the user has indicated a willingness to put up with the additional overhead.

Both Scheme and C allow the user to define polymorphic procedures. In Scheme polymorphism is unlimited, with type checking done dynamically. C functions can be made polymorphic through the use of type casts. This is limited to types having a common size, and also prevents any type checking on the polymorphic values. Pre-Scheme's static type checker, like that of ML and Haskell, correctly handles parametric polymorphism.

Pre-Scheme's module system is not part of standard Scheme but is inherited from Scheme 48 and described in [9].

The main differences between Scheme and Pre-Scheme are:

- A Pre-Scheme program's top-level forms are evaluated at compile time and may make use of full Scheme. This is exactly the evaluation that happens when a Scheme program is loaded into an interpreter.
- Pre-Scheme programs are statically typed by the compiler using a type reconstruction algorithm.
- Pre-Scheme has no garbage collection.
- In Pre-Scheme not all tail-recursive calls are done with proper tail-recursion. Proper tail recursion is guaranteed only for calls to `lambda` forms bound by `let` and `letrec` or when explicitly declared for a particular call.

Of course, since Pre-Scheme programs are Scheme programs, they can be developed and run using a Scheme implementation, in which case the programmer can make full use of the Scheme implementation's programming environment. This is especially helpful for storage management. A Pre-Scheme program can be developed and debugged in the presence of a garbage collector, with explicit storage management being added once the program is otherwise satisfactory.

The above differences are not enough on their own to get the desired program performance. Pre-Scheme presupposes a powerful compiler that does a fair amount of partial evaluation. Both the fancy compiler and the restrictions are necessary. Restrictions alone result in languages such as C and Pascal, which are efficient but not very powerful. While performance comparable to that of low-level languages is occasionally claimed for implementations of high-level languages, these claims are typically based on small benchmarks and not on full applications.

The current Pre-Scheme compiler produces C code and is based on the compiler described in [8, 7]. It has been used to compile the Scheme 48 virtual machine [9], a moderate-sized program that includes a byte-code interpreter and garbage collector. This particular application program, written in Pre-Scheme and compiled by the Pre-Scheme compiler, performs as well as comparable programs written directly in C. As shown in figure 1, when compiled with other systems, including Orbit [10], an optimizing Scheme compiler, it runs much more slowly. These timings are discussed in more detail in secion 6 below.

`Scheme->C`	120.00
Orbit, no in-lining	33.00
Orbit, in-lining	5.40
Pre-Scheme	0.84

Figure 1: Time, in seconds, for the Scheme 48 virtual machine to run `(fib 22)` (10^6 byte-code instructions) on a MIPS R3000, when compiled with various compilers.

2 How Pre-Scheme differs from Scheme

This section describes the differences between Scheme and Pre-Scheme in more detail.

2.1 Pre-Scheme is statically typed

Managing the type information needed for Scheme's dynamic type checking and type discrimination slows execution and denies the user direct access to machine data and instructions. For this reason Pre-Scheme is a statically typed language. Pre-Scheme's type system is a parametric polymorphic one similar to ML's. The main differences are in the handling of overloaded numeric operators and the use of an extended notion of polymorphism (see the section on type inference below).

The lack of dynamic type information means that type discrimination procedures such as `pair?` and `number?` are not available in Pre-Scheme. In the future we plan to add to Pre-Scheme record types, tagged unions, and tuples similar to those in ML. Currently the only data structures are those found in Scheme.

2.2 No garbage collection

Pre-Scheme's data model is that of Scheme's. Values are represented by references and arguments are passed by reference. Static typing allows the compiler to elide the references when the values in question are either atomic, such as numbers, or are known not to be side-affected.

The use of references means that values in general require some kind of allocated storage. There is no garbage collector, so any deallocation must be done by the programmer, using the system procedure `free`.

The lack of garbage collection makes closures less useful than in Scheme. However, in many instances code that would ordinarily require a closure can be compiled without one using various compiler optimizations. The compiler can be directed to indicate any code that will result in the creation of closures at run time.

2.3 Proper tail-recursion

Scheme implementations are required to be properly tail-recursive. There can be a significant run-time overhead for this on certain platforms. For example, when compiling into C implementing proper tail recursion involves the introduction of some form of driver loop. For Pre-Scheme the tail-recusion requirement only applies to calls to local procedures, defined as `lambda` forms that are bound by `let` or `letrec`. The programmer can declare that individual calls are to be compiled as properly tail-recursive. `(GOTO proc arg1 ...)` is syntax indicating that `proc` should here be called tail-recursively (assuming the `goto` form occurs in tail position).

2.4 Call-with-current-continuation

`call-with-current-continuation` is not available in Pre-Scheme. We plan to add downward continuations to Pre-Scheme in the future, since they incur no significant run-time cost.

2.5 Compile-time evaluation of top-level forms

A Pre-Scheme program's top-level forms are evaluated at compile time. This is identical to the evaluation that occurs when a Scheme program is loaded. After this evaluation the program consists of a sequence of definitions of procedures and literal values. The programmer must specify, at compile time, one or more entry points to the program.

Compile-time evaluation allows programmers to use all of Scheme in building and initializing complex data structures, which may include procedures, that the rest of the compilation process can treat as static. For example, if the programmer creates a vector of procedures at top-level, a call to an (unknown) element of the vector can be implemented as a computed-goto. All of Scheme is available for the evaluation of top-level forms; the restrictions described below do not apply.

3 Type reconstruction

The most complex restriction on Pre-Scheme programs is that they must be statically typed. The goal for Pre-Scheme's static type checking is to model Scheme's dynamic typing as accurately as possible while still allowing the compiler to decide upon a machine representation for every variable, to insert any necessary coercions, and to produce intelligible messages describing any type errors. Type reconstruction is done using a Hindley/Milner style polymorphic type reconstruction algorithm augmented to deal with overloaded operators and to insert coercion operations. Coercions are limited to those that are computationally inexpesive, for example between different numeric types. Coercions on procedural values are not done, because they would require dynamically allocating closures to hold the values and the code to do the necessary coercions. Type conflicts that can be repaired by the insertion of an unsafe coercion function are reported but do not stop the compilation process.

The type reconstruction algorithm produces a program augmented with coercions and a set of type relations encoding constraints on the coercions, similar to that in [12]. A representitive sample of the type inference rules are given in the appendix. Coercions, and their corresponding relations, are introduced wherever an expression produces a value. After type reconstruction is completed the compiler produces a solution to the type relations, and so determines the actual type of each coercion.

The type reconstruction algorithm allows for four different kinds of polymorphism:

- no polymorphism
- single size polymorphism. Different types of values are allowed, but they must all share a single representation size.
- multi-size polymorphism. Different copies of the procedure are required for different sizes of values.
- full polymorphism. A separate copy of the procedure's type, including any associated relations, is produced for each use. The procedure itself will be in-lined by the compiler.

If every value representation were the same size, single and multi-size polymorphism would be identical. Most ML implementations work in this fashion.

Full polymorphism is used when a procedure will be in-lined. As an example of full polymorphism consider the procedure `(define (add-one x) (+ x 1))`. Scheme's dynamic type checking and type discrimination mean that `add-one` can be used on any of Scheme's different numeric types, and its result will be coerced as necessary. To duplicate this statically requires using multiple copies of the `add-one` procedure, since different uses will require different coercions or different addition operators. The Haskell type system

```
(define (carefully op)
  (lambda (x y succ fail)
    (let ((z (op (extract-fixnum x) (extract-fixnum y))))
      (if (overflows? z)
          (goto fail x y)
          (goto succ (enter-fixnum z))))))

(define add-carefully (carefully +))

(define (arith op)
  (lambda (x y)
    (op x y return arithmetic-overflow)))

(define-primitive op/+ (number-> number->) (arith add-carefully))
```

Figure 2: Pre-Scheme code implementing the Scheme 48 virtual machine's addition instruction

allows the user to define overloaded procedures, but the overloading is resolved at run time, which is unacceptable for Pre-Scheme. Mitchell's algorithm for type inference with simple subtypes will introduce the coercions, but assigns a single type to `add-one` and would reject the following expression:

```
(* (add-one 1.5)
   (vector-ref v (add-one 3)))
```

`(add-one 1.5)` will force `add-one` to take a floating point argument and return a floating point result, which will cause a type error, since the result is passed to `vector-ref`, which requires an integer.

4 An Example

This section presents a code example to show that Pre-Scheme programs really are like Scheme programs, and not just C programs with Scheme syntax. The code is taken from the Scheme 48 virtual machine, which contains a byte-code interpreter, a garbage collector, and code for reading and writing heap images. This virtual machine is written entirely in Pre-Scheme. The example is the code implementing the virtual machine's addition instruction, which operates on small tagged integers.

The Scheme 48 virtual machine also serves as an example of utility of having a well-defined semantics for Pre-Scheme. The VLISP project [5] uses Scheme 48 as the basis for a fully verified Scheme implementation. Verifying the correctness of the Scheme 48 virtual machine would be much more difficult were portions of it written in C, as C's semantics are much more complicated, and less well defined, than Scheme's or Pre-Scheme's. As it was, the VLISP members were able to write a Pre-Scheme compiler that generated code for the Motorola 68000 and prove it correct [14].

Figure 2 illustrates the coding style used in the Scheme 48 virtual machine. The example consists of the code implementing the addition instruction. The procedure `carefully` takes an arithmetic operator and returns a procedure that performs that operation on two tagged arguments, either passing the tagged result to a success continuation, or passing the original arguments to a failure continuation if the operation overflows. `extract-fixnum` and `enter-fixnum` remove and add type tags to small integers. The function `overflows?` checks that its argument has enough unused bits for a type tag. `carefully` can then be used to define `add-carefully` which performs addition on integers.

`define-primitive` is a macro that expands into a call to the procedure `define-opcode` which actually defines the instruction. The three arguments to the macro are the instruction to define, input argument specifications, and the body of the instruction. The expanded code retrieves arguments from the stack, performs type checks and coercions, and executes the body of the instruction. This is a simple Scheme macro that would be painful, if not impossible, to write using C's limited macro facility.

5 Implementation

The current Pre-Scheme compiler is based on the transformational compiler described in [8, 7]. It uses the following optimization techniques:

- Beta reduction (substituting known values for variables).
- Block compilation. The entire program is compiled at once. This maximizes the opportunities for performing beta reduction. It also increases the number of programs that will be accepted by the type checker by allowing global dependency analysis.
- Transforming tail recursion to iteration. Tail recursive loops are identified and transformed into iterative ones as described in [8].
- Hoisting closures. Closures with no free lexically-bound variables are made into top-level procedures.
- Constant folding.
- C translation tricks. A number of transformations are applied to take advantage of the capabilities of the C compiler.

None of these techniques is new (for example, see [1, 10, 8, 2]), they have not previously been applied to a language intended for low-level programming. Pre-Scheme programs do not pay any penalty for with type tags, garbage collection, or full run-time polymorphism, since Pre-Scheme does not have them. As a result, the programmer can be given direct access to machine data. The best that can be done in a full Scheme implementation is to give the programmer these benefits within a single procedure, and often not even then. Not implementing Scheme in its full generality greatly increases the speed at which programs run the low-level expressiveness of the resulting language.

The Scheme 48 virtual machine illustrates the effectiveness of the Pre-Scheme compiler. Not including comments, the virtual machine consists of 570 forms containing 2314 lines of Scheme code, and is compiled to 13 C procedures contining 8467 lines of code.

Figure 3 shows the C code produced for the addition instruction. This is part of a large `switch` statement which performs instruction dispatch.

This code is not what we would have written if we had used C in the first place, but it is at least as efficient. The use of Pre-Scheme makes the program (moderately) comprehensible and easy to modify without incurring run-time cost. Figure 4 is the assembly code produced by GCC [16] from the above, for a MIPS R3000 processor.

Not surprisingly, the machine code closely follows the C code, since the C code is straightforward. The most important job done by the C compiler is register allocation.

While quite good, this code is not quite as good as we could have done had we started out writing in MIPS assembly language. For one thing, GCC did some unhelpful tail merging, which, while making the program a few instructions smaller,

```
case 46 : {
  long arg2_267X;
  RSstackS = (4 + RSstackS);          /* pop an operand from the stack */
  arg2_267X = *((long*)((unsigned char*)RSstackS));
  if ((0 == (3 & (arg2_267X | RSvalS)))) {       /* check operand tags */
    long x_268X;
    long z_269X;
    x_268X = RSvalS;
    z_269X = (arg2_267X >> 2) + (x_268X >> 2);  /* remove tags and add */
    if ((536870911 < z_269X)) {                       /* overflow check */
      goto L20950;}
    else {
      if ((z_269X < -536870912)) {                   /* underflow check */
        goto L20950;}
      else {
        RSvalS = (z_269X << 2);                 /* add tag and continue */
        goto START;}}
   L20950: {
    merged_arg1K0 = 0;
    merged_arg0K1 = arg2_267X;
    merged_arg0K2 = x_268X;
    goto raise_exception2;}}
  else {
    merged_arg1K0 = 0;
    merged_arg0K1 = arg2_267X;
    merged_arg0K2 = RSvalS;
    goto raise_exception2;}}
  break;
```

Figure 3: Compiler output for the Pre-Scheme code in figure 2[1]

resulted in the unnecessary jump at the end of the code in figure 4. Also, given that the tag for small integers is zero, we could possibly have added the two numbers with their tags intact and used a hardware overflow check instead of the two comparisons above (on a machine that provided such a check). These two failings could not been avoided by writing Scheme 48 in C instead of in Pre-Scheme. It is the C compiler that introduces the unnecessary jump, and C offers no direct access to the hardware overflow check.

6 Discussion

The timings in figure 1 show that the Pre-Scheme compiler does a much better job of compiling the Scheme 48 virtual machine than either Orbit or `Scheme->C`. Because the Scheme 48 virtual machine is written in a very modular fashion, with a number of procedure-based interfaces, it contains a large number of one and two line definitions.

[1]Many Scheme identifiers are not legal C identifiers, while on the other hand C is case-sensitive and Scheme is not. The compiler uses upper-case letters for the characters that are legal in identifiers in Scheme but not in C; for example `*val*` becomes `SvalS`. The compiler also introduces local variables to shadow global variables to improve register usage (similar to [17]). These introduced variables begin with `R`, thus `RSvalS` is a local variable shadowing the global variable `SvalS`.

```
$L450:  addu $19,$19,4      ; pop an argument from the stack
        lw   $6,0($19)
        or   $2,$6,$17      ; check tags on both arguments
        andi $2,$2,0x0003
        bne  $2,$0,$L1337
        li   $2,0x1fff0000 ; for overflow check
        ori  $2,$2,0xffff
        sra  $4,$6,2        ; remove tags
        sra  $3,$17,2
        addu $4,$4,$3       ; do the addition
        slt  $2,$2,$4
        bne  $2,$0,$L492    ; check for overflow
        move $7,$17
        li   $2,-536870912 ; check for underflow
        slt  $2,$4,$2
        beq  $2,$0,$L1619
        j    $L1670
        move $22,$0
$L1619: j    $L221          ; jump to instruction dispatch
        sll  $17,$4,2       ; add tag
```

Figure 4: Assembly code for the C code in figure 3

If these procedures are not compiled in-line the system's performance is very poor, as shown in the first two timings. In-lining these definitions with Orbit gives much better performance, but still not that of the Pre-Scheme compiler. In both cases Orbit was in-lining standard Scheme procedures and using fixnum-specific arithmetic. In-lining was not done with `Scheme->C` as its mechanism for handling user in-lining declarations was not sufficiently robust.

One reason that Orbit's output is so much slower is that Orbit does not do top-level form evaluation, and as a result compiles the code for each of the virtual machine's op-codes as a separate top-level procedure and the exectution of every instruction then requires a full procedure call. This cost could be avoided rewriting the virtual machine as a single large `case` or `cond` expression, at the cost of convoluting the code. Other costs are more fundamental, such as the need to maintain type information at run time.

7 Related Work

In this section we compare Pre-Scheme with three other approaches: the design of standard languages, high-performance Scheme and Lisp implementations, and low-level languages with Lisp syntax.

7.1 C, Pascal, and other low-level languages

These languages use syntactic restrictions to force programmers to write programs that can be run efficiently using a particular type of implementation. Syntactic restrictions have the advantage of being easy to understand and easy to enforce. Unfortunately, implementation limitations tend not to be easily modelled in syntax, with the result that the syntactic restrictions are much stronger than necessary. A good example of this is how closures are avoided in C and Pascal. Closures result from allowing the unrestricted use of nested procedures as values. C allows the unrestricted use of procedures as values, but does not allow nested procedure declarations. Pascal allows nested procedures, but restricts how procedure values may be used. Enforcing exactly the implementation's restriction would require syntactically distinguishing procedures that contained free lexical references from those that don't.

7.2 Other languages with Lisp syntax

Lisp syntax has been used for a variety of low-level languages, from assembly code (LAP code in many Lisp implementations) to FORTRAN (the misnamed Tinylisp used in [4]). This both allows the use of syntactic macros and makes parsing the language trivial. Other than the syntax, these languages typically have little or nothing to do with Lisp, or with high-level languages in general.

7.3 High-performance implementations

Most efforts to produce efficient Scheme or Lisp implementations are hampered by being required to implement the full language. Low-level language speed has been claimed for particular Lisp and Scheme implementations, usually on the basis of running some fairly small benchmarks [10, 13]. Here, instead of a small benchmark we have a non-trivial, useful application, written in Pre-Scheme, that performs as well as similar programs written in C. Scheme 48, when the VM is compiled using the Pre-Scheme compiler, runs at about the same speed of SCM4, a widely used Scheme implementation hand-written in C (the two have very different implementation strategies and the actual relative speeds varies widely depending on the code being run). As discussed in section 6, Scheme 48's perforance decreases drastically when its VM is compiled using the Orbit compiler.

Like the Pre-Scheme compiler, [3] and [17] translate high-level languages (Scheme and ML) into C. `Scheme->C` translates calls to global Scheme procedures into calls to C procedures (as does the Pre-Scheme compiler for most calls), and thus has global tail recursion only if it is implemented by the C implementation. `CMU-ML->C` uses a driver loop to implement global tail recursion in C (as the Pre-Scheme compiler does when the user declares that a call should be tail-recursive). Unlike Pre-Scheme, `Scheme->C` uses type bits and both `->C` implementations require that programs be linked to a special run-time library. `Scheme->C` also does not do the global optimizations necessary to get maximal performance.

8 Future Work

The Pre-Scheme language could be enhanced by adding additional functionality to the compiler, allowing a larger set of Scheme programs to be compiled. Downward closures could be allowed, as they are in Pascal, since doing so would not compromise the efficiency of compiled programs. This might require declarations on the part of the programmer to indicate when a closure could be passed downwards, instead of needing to be eliminated by beta reduction. Downward continuations could also be implemented using C longjumps.

Obtaining maximally efficient code from the current compiler requires programmer directives to make up for the compiler not having information about the dynamic behavior of the program or knowledge of the target architecture. For example, the programmer may want a procedure that deals with an exceptional case not to be inlined, since it might slow down the normal case, or desire that within a given set of procedures a particular global variable be shadowed by a local variable, and thus end up in a machine register. This opens up two avenues for further design and experimentation. A more sophisticated compiler could get by with less user information, and more sophisticated directives could produce better C output and increase the number of programs that could be compiled effectively.

9 Conclusion

Pre-Scheme is Scheme restricted to those programs that can be compiled to very efficient object code using current techniques.

Pre-Scheme programs can be run as Scheme programs or compiled into native code. Running them as Scheme programs gives full access to the debugging and automatic storage reclamation features of the Scheme implementation. But unlike Scheme programs, Pre-Scheme programs can be statically type-checked and compiled into native code that does not require type tags, garbage collection or other features that slow execution. Pre-Scheme programs' use of these Scheme features is restricted to exactly those instances that can be compiled efficiently.

We have shown that these restrictions allow the use of many of Scheme's powerful features in writing low-level programs, without sacrificing perfomance.

10 Acknowledgements

Pre-Scheme was developed as part of the Scheme 48 project, which is a joint effort on the part of Jonathan Rees and myself. Jonthan Rees, Suresh Jaganathan, Rick Mohr, and Mitch Wand provided many helpful comments on this paper.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, Cambridge, 1992.

[3] J. Bartlett. Scheme→c: A portable scheme-to-c compiler. Technical report, DEC Western Research Laboratory, 1989.

[4] John Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1985.

[5] J. D. Guttman, L. G. Monk, J. D. Ramsdell, W. M. Farmer, and V. Swarup. A guide to VLISP, a verified programming language implementation. Technical Report M92B091, The MITRE Corporation, 1992.

[6] P. Hudak and P. Wadler (eds.). Report on the programming language haskell. Technical Report YALEU/DCS/TR-777, Department of Computer Science, Yale University, New Haven, CT, 1990.

[7] Richard Kelsey. Compilation by program transformation. Technical Report YALEU/DCS/TR-702, Department of Computer Science, Yale University, New Haven, CT, 1989.

[8] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Conf. Rec. 16 ACM Symposium on Principles of Programming Languages*, pages 281–292, 1989.

[9] Richard Kelsey and Jonathan Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7:315–335, 1994.

[10] David A. Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams. Orbit: An optimizing compiler for scheme. In *Proceedings SIGPLAN '86 Symposium on Compiler Construction*, 1986. *SIGPLAN Notices 21*(7), July, 1986, 219-223.

[11] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[12] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1:245–285, 1991.

[13] A. Nagasaka, Y. Shintani, and T. Ito. Tachyon common lisp: An efficient and portable implementation of cltl2. In *Proc. 1992 ACM Conf. on Lisp and Functional Programming*, 1992.

[14] D. P. Oliva, Ramsdell, and M J. D., Wand. The vlisp verified prescheme compiler. *Lisp and Symbolic Computation*, 8(1 & 2):111–182, 1995.

[15] Jonathan A. Rees and eds. Clinger, William C. Revised[3] report on the algorithmic language scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[16] R. Stallman. *Using and Porting GNU CC.* Free Software Foundation, 1989.

[17] D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling Standard ML to C. Technical report, School of Computer Science, Carnegie Mellon University, 1991.

Appendix: type reconstruction rules

Simplified Scheme Syntax

N	`literal constant`
`(lambda (`I`) `E`)`	`procedure`
`(let (`I E_{value}`) `E_{body}`)`	`local binding`
I	`identifier`
`(set! `I E`)`	`variable binding mutation`
`(`E_{proc} E_{arg}`)`	`procedure application`
`(if `E_{test} E_{cons} E_{alt}`)`	`conditional`

The statement:

$$A \vdash E \to E' : t$$

means that in type environment A expression E expands to E', which has type t.

$$\frac{A[I \mapsto s] \vdash E \Rightarrow E' : t}{A \vdash \texttt{(lambda (}I\texttt{) }E\texttt{)} \Rightarrow \texttt{(lambda (}I\texttt{) }E'\texttt{)} : s \to t}$$

$$\frac{\begin{array}{l} A \vdash E_1 \Rightarrow {E_1}' : s \\ A[I \mapsto \mathit{Oracle}(A, {E_1}', s)] \vdash E_2 \Rightarrow {E_2}' : t \end{array}}{A \vdash \texttt{(let (}I\ E_1\texttt{)}\ E_2\texttt{)} \Rightarrow \texttt{(let (}I\ {E_1}'\texttt{)}\ {E_2}'\texttt{)} : t}$$

The oracle is needed to choose between the various kinds of polymorphism.

$$\frac{\mathit{integer} \sqsubseteq t}{A \vdash N \Rightarrow \texttt{(coerce-integer->}t\ N\texttt{)} : t}$$

$$\frac{t \sqsubseteq t'}{A[I \mapsto t] \vdash I \Rightarrow \texttt{(coerce-}t\texttt{->}t'\ \texttt{I)} : t'}$$

$$\frac{A[I \mapsto t] \vdash E \Rightarrow E' : t}{A[I \mapsto t] \vdash \texttt{(set! }I\ E\texttt{)} \Rightarrow \texttt{(set! }I\ E'\texttt{)} : \mathit{unit}}$$

$$\frac{\begin{array}{l} A \vdash E_1 \Rightarrow {E_1}' : s \to t \\ A \vdash E_2 \Rightarrow {E_2}' : s \\ t \sqsubseteq t' \end{array}}{A \vdash \texttt{(}E_1\ E_2\texttt{)} \Rightarrow \texttt{(coerce-}t\texttt{->}t'\ \texttt{(}{E_1}'\ {E_2}'\texttt{))} : t'}$$

$$\frac{\begin{array}{l} A \vdash E_1 \Rightarrow {E_1}' : \mathit{boolean} \\ A \vdash E_2 \Rightarrow {E_2}' : t \\ A \vdash E_3 \Rightarrow {E_3}' : t \\ t \sqsubseteq t' \end{array}}{\begin{array}{l} A \vdash \texttt{(if }E_1\ E_2\ E_3\texttt{)} \\ \quad \Rightarrow \texttt{(coerce-}t\texttt{->}t'\ \texttt{(if }{E_1}'\ {E_2}'\ {E_3}'\texttt{))} : t' \end{array}}$$