On the Use of LISP in Implementing Denotational Semantics

A Progress Report

Peter Lee and Uwe Pleban
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor

Abstract

Automatic compiler generators and semantics systems typically produce compilers which depend heavily on the mechanism of β-reduction. This is particularly true of those systems based on denotational semantics, since their descriptive notations are based on the λ-calculus.

Performing β-reductions is expensive, however, and this is a primary reason for the extreme inefficiency of automatically-generated compilers and the code they produce. Since LISP is in many respects similar to the λ-calculus, it seems a reasonable idea to generate compilers which rely on the LISP EVAL function rather than a β-reducer. Then, the expensive simulation of β-reductions is avoided, and the efficiency of LISP is obtained. Unfortunately, moving to LISP is complicated by the fact that the generated compilers quite often depend on a β-reducer's ability to *partially* evaluate an expression — a capability lacking in LISP.

We have implemented a compiler generator called MESS which produces realistic and efficient compilers written in SCHEME. MESS processes *modular denotational descriptions*, and exploits this modularity in order to avoid the dependence on partial evaluation. An added benefit of our approach is that the output of the generated compilers can be directly processed by an automatically-generated table-driven code generator. This makes it possible to obtain object code which compares favorably with that produced by hand-crafted compilers.

Our results thus far are quite encouraging, as we are finding that the compilers generated by MESS are significantly more efficient and realistic than those produced by other systems.

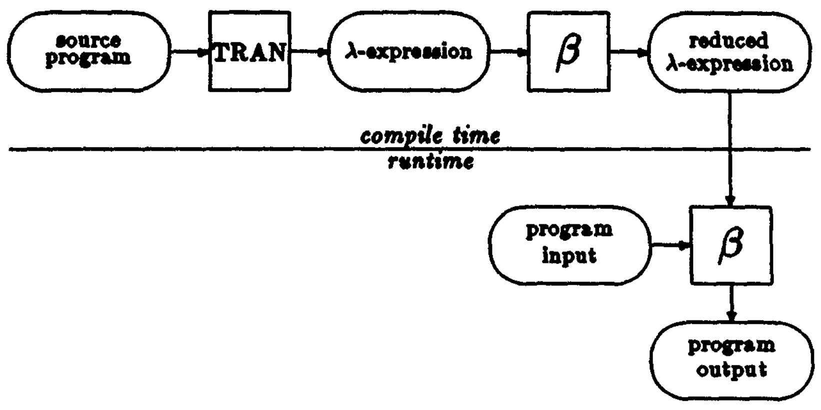
Introduction

Several compiler generators and semantics systems have been developed [Mos79,Pau82,Wan84] based on the direct implementation of denotational semantics [Sto77]. The compilers generated by these systems work in a rather strange way. Given a source program, they typically derive its λ-expression "meaning" by performing a syntax-directed translation based on the semantic equations. This λ-expression is then usually simplified, through β-reduction, as much as possible at compile time. The resultant λ-expression is taken as the object code and can be "executed" by performing further β-reductions in the presence of an input file.

Unfortunately, these compilers are not very practical. This is due in large part to their inefficiency, which stems from the expense of performing β-reductions. Since LISP is in many respects similar to the λ-calculus (particularly lexically-scoped, full-funarg LISP dialects such as SCHEME [StS78]), it seems reasonable to allow the generated compilers to derive LISP programs instead of λ-expressions. This would allow the use of a LISP implementation's compiler or EVAL function in executing the "object" code. Then, the explicit simulation of β-reductions is avoided, and the efficiency of LISP is obtained.

The problem with this approach is that LISP programs are evaluated in one fell swoop — there is no concept of "partial" evaluation as there is when considering the β-reduction of λ-expressions. This is particularly crippling at compile time, when certain runtime entities, such as the program input file, may be "unbound." Since most LISP implementations treat the evaluation of unbound entities as fatal errors, a generated compiler based on LISP would often fail during compilation. On the other hand, a β-reduction mechanism in the same situation simply stops evaluation, and returns the partially-reduced expression as the "answer." Furthermore, this "answer" expression has the desirable property that *only* runtime computations are left to do — all compile time computations have at this point been reduced away.

In order for compiler generation based on denotational semantics to become feasible for realistic and useful languages, we believe the generated



This figure shows the typical process of compilation by an automatically-generated compiler. The source program is first translated to its λ-expression meaning, and then β-reduced as far as possible. This results in a reduced λ-expression which is taken to be the object code for the program. The program can then be run by performing further reductions in the presence of the input file.
A significant amount of time and storage is consumed by the β-reduction processes, β, and thus it is desirable to replace these by a more efficient mechanism, such as a SCHEME evaluator.

Figure 1: Semantics-Based Compilation

compilers must accomplish the following:

1. Avoid explicit β-reduction while still retaining the partial evaluation property.[1]
2. Avoid the introduction of unnecessary closures into the object code.

For languages like Pascal, closures in the object code are always unnecessary since a simple stack discipline may be used to represent environments. Other languages, for instance SCHEME, which have higher-order functions and continuations as first-class objects, may require closures; but in this case they should be used only to handle these particular language features. Unfortunately, automatically-generated compilers usually introduce many unnecessary closures, e.g., to represent the program store, into the object code.

We say that a compiler which accomplishes the two goals listed above is *realistic*. Note that by this definition, hand-crafted compilers are realistic.

We have developed a new technique for language specification based on modular denotational descriptions. The technique has been implemented in a compiler generation system called MESS[2] which produces realistic compilers written in SCHEME. This paper describes MESS and its use of SCHEME in generating efficient and realistic compilers.

[1]A compiler without the partial evaluation property has no "*pep*." This notion of partial evaluation should not be confused with that in Jones' "mix" operator [Jon84].

[2]*M*arvelous *E*xtensible *S*emantics *S*ystem.

Semantics-Based Compiler Generation

Figure 1 shows the compilation process of compilers generated by "classical" systems such as Mosses' Semantics Implementation System (SIS) [Mos79] and Paulson's Semantics Processor (PSP) [Pau82]. Our extensive experience with these systems has already been presented elsewhere [BoB82,Ple84a], so we shall refrain from discussing them in detail here.

Instead, we simply point out that a significant amount of time and storage is consumed by the β-reduction processes (the boxes marked "β"). This is because the reductions are performed on program graphs and involve a considerable amount of copying of substructures. It is still an open question whether this can be done efficiently.[3] Indeed, one reason for SCHEME's efficiency is the fact that lambda-variables can be "compiled away" into stack offsets. A β-reducer, on the other hand, can not do this since it must allow for partial evaluation.

Thus, a more promising strategy is to derive LISP (i.e., SCHEME) expressions instead of λ-expressions, and then use a LISP implementation for the reductions.

[3]Paulson's system compiles the reduced λ-expressions to code for an SECD machine [Lan64] for greater efficiency. However, the compile time reductions are still performed by expanding and simplifying the program graph.

SCHEME as the Target Language

This is the method taken by Wand's Semantic Prototyping System (WaSP) [Wan84]. WaSP-generated compilers translate a source program into a semantically equivalent SCHEME target program. The application of the target program to the required runtime information (such as the input file and the initial store) evaluates to the program's output.

However, since there is no concept of partial evaluation in SCHEME, WaSP-generated compilers do not perform any reductions at compile time. Instead, all reductions are deferred to runtime. While this approach has the advantage of being able to use the highly efficient SCHEME implementation, the generated compilers are far from realistic. Compile time computations, e.g., for static semantic checking and storage allocation, are embedded in the derived SCHEME programs, and must be performed *every time the program is executed*, thus negating much of the advantage of using SCHEME.

A New Technique for Semantic Specification

In order to regain the partial evaluability property, a compiler generation system must be able to distinguish between those portions of the semantic specification describing static components of the language, and those describing dynamic components. Then, runtime entities in the semantics could be "marked" so as to avoid their evaluation at compile time.

MESS is able to do this by enforcing a *modularity* in the semantic specifications. In a modular semantics, the semantic equations are not written in a low-level λ-notation; rather, a higher-level notation is used which allows one to abstract away from model-dependent details such as stores and continuations. This concept of modularity is the same as that proposed by Mosses in his *abstract semantic algebras* [Mos84].

A complete description of our specification technique is given in [Lee86]. Here we give just a few brief examples to highlight the main ideas. Consider the following fragment of a semantic specification as might appear in one of the classical systems:[4]

```
C [[stmt1 ";" stmt2]] env store =
   C [[stmt2]] env
      ( C [[stmt1]] env store ) ;

E [[expr1 "+" expr2]] env store =
   (E [[expr1]] env store) +
   (E [[expr2]] env store) ;
```

[4]In this paper, all fragments of semantic specifications are written in the embellished, ML-like [Mil85] notation used by MESS.

These equations define the semantics of statement sequencing and addition expressions in the so-called "direct" style. The semantic variable env represents the static environment, and store represents the program store or state. These equations are non-modular because model-dependent details such as stores are intertwined with the language semantics. Thus, if a change is made to the semantic model, drastic changes must be made to the semantic equations. For example, if the model changes so that continuations become necessary (e.g., to model escapes from loops), then the equations must be rewritten as:

```
C [[stmt1 ";" stmt2]] env cont =
   C [[stmt1]] env
      { C [[stmt2]] env cont } ;

E [[expr1 "+" expr2]] env kont =
   E [[expr1]] env { fn e1.
      E [[expr2]] env { fn e2.
         kont (e1 + e2) } } ;
```

which is *completely* different from the previous equations. In these equations, cont and kont represent command and expression continuations, respectively.

In MESS, the semantics is written in a modular fashion:

```
C [[stmt1 ";" stmt2]] env =
  seq (C [[stmt1]] env, C [[stmt2]] env) ;

E [[expr1 "+" expr2]] env =
  add (E [[expr1]] env, E [[expr2]] env) ;
```

where the model-dependent details have been encapsulated in the definitions of the semantic "operators" seq and add. These operators produce values in the *action domains* A_I (imperative actions) and A_V (value-producing actions), and can be defined in any number of ways without affecting these equations. For example, a continuation-style definition can be given as follows:

```
ImpAction = CONT -> CONT ;
ValAction = KONT -> CONT ;

seq : ImpAction * ImpAction -> ImpAction ;
seq (c1, c2) =
   fn cont. c1 { c2 cont } ;

add : ValAction * ValAction -> ValAction ;
add (e1, e2) =
   fn kont. e1 { fn ev1.
      e2 { fn ev2.
         kont (ev1 + ev2) } } ;
```

Other definitions of the operators are possible, for example a direct-style definition:

```
ImpAction = STORE -> STORE ;
ValAction = STORE -> EV ;

seq : ImpAction * ImpAction -> ImpAction ;
seq (c1, c2) =
   fn store. c2 ( c1 store ) ;

add : ValAction * ValAction -> ValAction ;
add (e1, e2) =
   fn store.
      ( e1 store ) + ( e2 store );
```

or even a SCHEME program:

```
(define (seq c1 c2)
  (begin c1 c2))

(define (add e1 e2)
  (+ e1 e2))
```

The key point is that *modularity preserves the ability to choose any form of definition, or implementation, of the semantic model, whereas it is destroyed in standard denotational descriptions by the intertwining of the language semantics with the semantic model.*

In addition to separating the semantics from the semantic model, we believe it is important to distinguish operators which represent dynamic language concepts from those which represent static concepts. A_I and A_V are clearly dynamic (i.e., runtime) action domains, whereas operators in the domain of environment-producing actions, A_D, might be static actions. One can regard static actions as representing compile time computations, and dynamic actions as pieces of object code. This separation of static and dynamic action domains not only increases the comprehensibility of the semantic descriptions, but also allows MESS to decide what operations can or should be performed at compile time.

We use the term *macrosemantics*, or simply *semantics*, to refer to a modular semantic specification which completely avoids explicit references to model details. The definition of the runtime operators, then, is called the *microsemantics*. This terminology is analogous with the terms *microsyntax* and *syntax* used for describing elements of language syntax.

MESSy Compiler Generation

The basic idea of MESS is that modularity in the semantic specifications is *enforced* by the system. This allows MESS to decide which portions of the semantics should be evaluated at compile time, and which should be deferred to runtime.

Figure 2 shows the overall structure of MESS. MESS has been implemented on an IBM Personal Computer, with the front-end generator written entirely in Pascal, and the semantic analyser in SCHEME. The Pascal implementation used is Turbo Pascal [Bor85], and the SCHEME implementation is TI PC SCHEME [TI85].

As an example, Appendix A gives a macrosemantic specification (corresponding to "Ma spec." in figure 2) in MESS format for the language KleinPL. KleinPL is an imperative language with arrays, non-recursive procedures, and the usual Pascal-like control structures.

MESS uses a semantic metalanguage similar to ML [Mil85], i.e., it is applicative and has polymorphic types. The `interface` declaration in the example specification tells MESS which file contains the specification of the abstract syntax ("AS spec."). The abstract syntax specification is generated automatically by a compiler front-end generator, and is used by MESS to ensure the consistency of the abstract syntax expressions appearing in the macrosemantic specification with those specified in the front-end specification ("FE spec.").

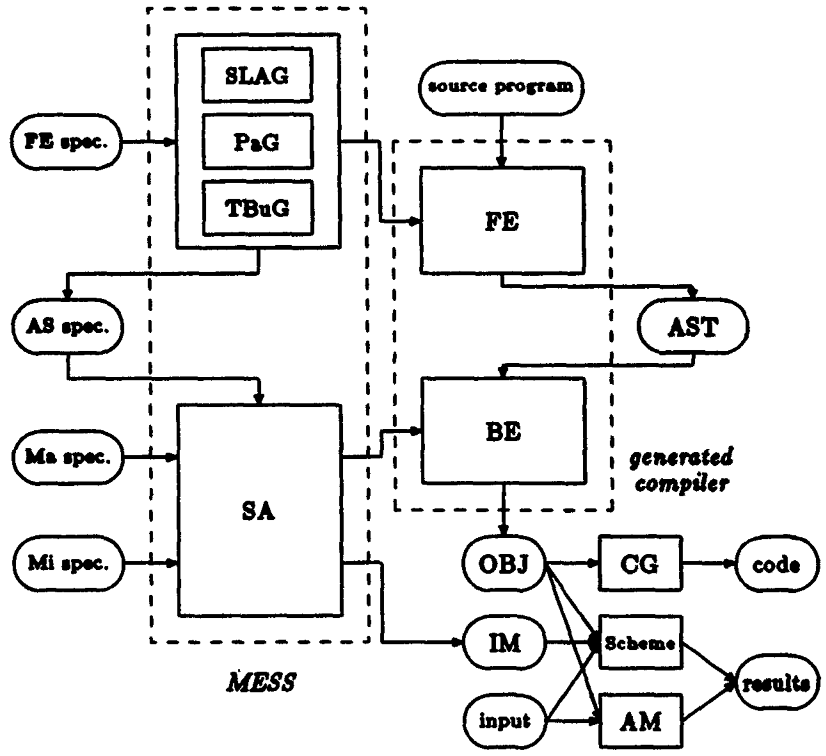
The `microsemantics` declaration gives the name of a file containing the specification of the microsemantics (corresponding to "Mi spec." in figure 2), i.e., the definitions of the dynamic operators and action domains. An example of a microsemantic specification is given in Appendix B. This microsemantic specification is converted by MESS into a SCHEME program which implements the operators ("IM"). This program can then be used as an environment in which compiled KleinPL programs can be executed.

A KleinPL Compiler

The compiler generated by MESS ("FE" and "BE" in figure 2) from the KleinPL specifications ("FE spec." and "Ma spec.") is a syntax-directed transducer which translates KleinPL source programs into SCHEME code. The macrosemantic specification is 434 (well-commented) lines long, and required approximately 8 man-hours to write and debug. The microsemantic specification is 389 lines long, and required about 18 hours of work. In both cases, much of the time was spent fixing type errors, since the MESS type checker is not yet complete.

The generation of the front-end requires 52.47 seconds,[5] and results in a 5,000 line Turbo Pascal program which performs lexical analysis and parsing of KleinPL programs. Most of the Pascal code is for automatic syntactic error recovery. Note that the front-end generator is still under development, and we expect its running time to improve considerably. Semantic analysis and back-end generation requires 180.81 seconds, and results in a

[5]All timings were taken on an IBM PC with an "accelerated" 10MHz clock.



This pictorial representation of *MESS* shows the various phases of compiler generation. The semanticist provides specifications for the front-end, macrosemantics, and microsemantics (FE spec., Ma spec., and Mi spec., respectively). A specification of the abstract syntax (AS spec.) may also be given, although the front-end generator, consisting of the Simple Lexical Analyser Generator, Parser Generator, and Tree-Builder Generator (SLAG, PaG, and TBuG) generates this automatically. The semantics analyser (SA) analyses the semantic descriptions and produces the compiler back-end (BE) and a SCHEME implementation of the microsemantic operators (IM). The back-end transforms abstract syntax trees (generated by the front-end (FE) or manually written) into object code (OBJ). If a code generator (CG) is available, the target code can be translated to machine code. Otherwise, either an abstract machine (AM) or the implementation of the microsemantics (IM) may be used to execute the program.

Figure 2: Our big *MESS*.

868 line (pretty-printed) SCHEME program. Excerpts of this program are given in Appendix C.

The translation of the KleinPL macrosemantics to the SCHEME program in Appendix C is relatively straightforward. The SCHEME program translates abstract syntax trees into prefix expressions, where the prefix operators are taken from the microsemantics. Note that in the SCHEME equivalent of the KleinPL macrosemantics the names of the microsemantic operators are quoted – it is this quoting which prevents evaluation of these runtime operators at compile time. Since the quoting is explicit, there is no need to depend on partial evaluation. Other pieces of the code involved only with compile time computations are not quoted, and thus are evaluated at compile time, which is the desired effect.

Appendix D gives a bubble-sort program written in KleinPL, and excerpts of the object code produced for it by the MESS generated compiler.

The compilation requires 20.26 seconds, and results in 308 lines of object code. The compiler spends its time as follows:

lexical analysis and parsing	2.41 sec.
abstract syntax tree building	8.62 sec.
translation to object code	2.47 sec.
object code output	6.76 sec.

Executing the Object Code

The object code is a prefix-form expression comprised solely of applications of microsemantic operators. Thus, this expression can be evaluated by the SCHEME system in an environment which has been augmented by an implementation of the microsemantics. This implementation can be obtained in a number of ways.

First, MESS can automatically produce, from a microsemantic specification, a SCHEME program implementing the microsemantics ("IM" in figure 2). Appendix E gives fragments of the SCHEME program derived for the continuation microsemantics given in Appendix B. MESS requires 154.12 seconds to generate this program, which is 497 (pretty-printed) lines long. We have also written a microsemantics for these operators in the "direct" style. For this specification, which is 364 lines long, MESS requires 142.20 seconds to generate the implementation, resulting in a 476 line SCHEME program.

Alternatively, one can write an "abstract machine" implementation of the microsemantics by hand ("AM" in figure 2). In this case, one might take advantage of special knowledge about the semantic model, for example that the store can be implemented as a large vector, in order to gain execution time efficiency. We have written an abstract machine which implements the operators specified in Appendix B as a 138 line SCHEME program.

Finally, since the object code is in prefix form and the operators are suitably low-level, one can use a table-driven, Bird-style code generator [Bir82] in order to generate machine code. We have written such a code generator for the Intel 8086 machine (this is the CPU used in the IBM PC), resulting in a code generator written in PROLOG. The PROLOG implementation used is Turbo Prolog [Bor86]. For the bubble-sort program, the code generator produces 205 instructions requiring approximately 400 bytes of storage.

The following table gives the execution times for the bubble-sort program, given a worst-case input of ten integers, in each of the microsemantic implementations described above.

continuation-style	40.59 sec.
direct-style	14.94 sec.
abstract machine	6.75 sec.
8086 machine code	< .50 sec.

Comparison with PSP

We are currently in the process of porting Paulson's Semantics Processor (PSP) [Pau82] to the IBM PC in order to provide for a direct comparison of execution times. At the time of this writing, enough of PSP has been ported to allow us to generate a small compiler. However, the PSP-generated compilers produce code for an SECD machine [Lan64] which we have not yet finished porting. Thus, we can compare only compiler generation times and compile times.

For our test language we take ToyPL, which is a very small imperative language with arrays. The time required by each system (discounting I/O overhead time) to generate a ToyPL compiler is given in the following table:

MESS	PSP
150.78 sec.	28.00 sec.

Part of the large time difference between MESS and PSP can be attributed to the (presently) slow front-end generator in MESS. However, MESS is also spending considerably more time in semantic analysis and back-end generation as well.

For a small ToyPL program, the running times (again discounting I/O overhead time) for these compilers are as follows:

MESS	PSP
4.61 sec.	17.20 sec.

The significant speed advantage exhibited by the MESS-generated ToyPL compiler can be attributed primarily to its use of the SCHEME

EVAL function for compile time reductions. The PSP-generated compiler, on the other hand, is hampered by the slow β-reduction process.

Making a MESS

Although MESS is now completely operational, some implementation work still remains. The type-checker for the semantics analyser is incomplete, and we have yet to write larger, more realistic microsemantic specifications. Our plan is to provide a microsemantics library, complete with abstract machines and code generators, which define operator sets rich enough to handle a Pascal semantics. We believe that this will provide the semanticist with a solid basis on which to experiment with language design and compiler generation.

Conclusion

This paper has described a compiler generator called MESS which is able to automatically derive realistic compilers from formal semantic descriptions. As an example, we showed how MESS generates a compiler for KleinPL, a language with control structures, procedures, and arrays. The generated compiler is both efficient and realistic. We are not aware of any other system which is able to generate such compilers from formal specification. Furthermore, we believe the engineering feasibility of our approach is amply demonstrated by the fact that MESS is operational on a desktop microcomputer.

Sethi has developed a system [Set81] which can generate a compiler which produces machine code for a language with all of the control structures in the C language. Unfortunately, the method used in his system does not work for procedures and data structures. Also, Appel has recently described a new compiler generator [App85]. Although the published accounts of his work are still quite preliminary, the specification technique appears to be rather *ad hoc*. Furthermore, the reported compile time of one VAX CPU second per line of code is, we believe, much too slow to be considered realistic.

The compilers generated by MESS are written in SCHEME, which is used as a highly efficient λ-calculus machine. Our experience indicates that SCHEME is a particularly good language for both compiler writing and generation. This should not be surprising, as compilers quite often deal with tree structures which can be manipulated easily and efficiently in SCHEME. In addition, SCHEME handles higher-order functions (i.e., full funargs) and continuations as first-class objects — both features find good use in automatic compiler generation. Finally, SCHEME implementations usually come with compilers which are quite good at optimisation, and this helps to reduce the running time of the generated compilers.

References

[App85] Appel, A. C. Semantics-directed code generation. In Conf. Rec., 12th Ann. ACM Symp. Principles Programming Languages, New Orleans, Louisiana, Jan. 1985, 315-324.

[Bir82] Bird, P. An implementation of a code generator specification language for table driven code generators. In Proc. SIGPLAN '82 Symp. Compiler Construction, *SIGPLAN Notices 17*, 6, (June 1982), 44-55.

[BoB82] Bodwin, J., Bradley, L., Kanda, K., Litle, D., and Pleban, U. Experience with an experimental compiler generator based on denotational semantics. In Proc. SIGPLAN '82 Symp. Compiler Construction, *SIGPLAN Notices 17*, 6, (June 1982), 216-229.

[Bor85] *Turbo Pascal Reference Manual (version 3.0)*. Borland International, Inc., 1985.

[Bor86] *Turbo Prolog Owners Handbook*. Borland International, Inc., 1986.

[Jon84] Jones, N. D., Sestoft, P., and Sondergaard, H. An experiment in partial evaluation: The generation of a compiler generator. Tech. Rep., Institute of Datalogy, University of Copenhagen, 1984.

[Lan64] Landin, P. J. The mechanical evaluation of expressions. *The Computer Journal 6*, (1964), 308-320.

[Lee86] Lee, P. The Automatic Generation of Realistic Compilers From High-Level Semantic Descriptions. Ph.D. Thesis, Dep. of Electrical Engineering and Computer Science, Univ. of Michigan, Ann Arbor, forthcoming.

[Mil85] Milner, R. The standard ML core language. *Polymorphism II*, 2, (Oct. 1985).

[MiS76] Milne, R. E., and Strachey, C. *A theory of programming language semantics*. Chapman and Hall, London, 1976.

[Mos79] Mosses, P. SIS — Semantics implementation system. Tech. Rep. DAIMI MD-30, Computer Science Dep., Aarhus Univ., Aug. 1979.

[Mos84] Mosses, P. A basic abstract semantic algebra. In *Lecture Notes in Computer Science*, vol. 173: *Semantics of Data Types*, Springer-Verlag, Berlin, 1984, 87-107.

[Pau82] Paulson, L. A semantics-directed compiler generator. In Conf. Rec., 9th Ann. ACM Symp. Principles Programming Languages, Albuquerque, New Mexico, Jan. 1982, 224-239.

[Ple84a] Pleban, U. F. Compiler prototyping using formal semantics. In Proc. SIGPLAN '84 Symp. Compiler Construction, *SIGPLAN Notices 19*, 6, (June 1984), 94-105.

[Set81] Sethi, R. Control flow aspects of semantics directed compiling. Tech. Rep. 98, Bell Labs., 1981; also in Proc. SIGPLAN '82 Symp. Compiler Construction, *SIGPLAN Notices 17*, 6, (June 1982), 245-260.

[Sto77] Stoy, J.*Denotational semantics: The Scott-Strachey approach to programming language theory*. MIT Press, Cambridge, 1977.

[StS78] Steele, G. L., and Sussman, G. J. The revised report on SCHEME, a dialect of LISP. MIT AI Memo 452, Cambridge, 1978.

[TI85] *TI Scheme Language Reference Manual*, Texas Instruments, Inc., 1985.

[Wan84] Wand, M. A semantic prototyping system. In Proc. SIGPLAN '84 Symp. Compiler Construction, *SIGPLAN Notices 19*, 6, (June 1984), 213-221.

Appendix A: The KleinPL Macrosemantics (excerpts)

```
semantics KleinPL;

interface      "KleinPL";
microsemantics "cont";

semantic domains

   (* ENV: Static environments.
    * Used to keep track of the statically-determined information (the MODE) for each identifier.
    * Also contains a "new" location pointer for storage allocation and the current block nesting level. *)
   ENV = (IDENT -> MODE) * LOC * INT;

   (* IDENT: Identifiers in the static environment.
    * These are represented as tuples:  (AST leaf giving the id name, its block level) *)
   IDENT = AST * INT;

   (* MODE: Identifier modes.
    * For each of the three denotations for identifiers, we keep track of its address (or
    * its parameter's address, in addition to other information. *)
   TYPE = union int_type | bool_type;
   MODE = union none | var of (LOC * TYPE) | array of (LOC * UPPERBOUND * TYPE) | procedure of (LOC * TOKEN);

   (* DECL_ELEMENT: Declaration elements.
    * These can be either a variable, which has a name; or else an array with a name and size. *)
   DECL_ELEMENT = union varDecl of AST | arrayDecl of (AST * AST);

auxiliary functions

   initEnv : ENV = ((fn id. none), 0, 0);

   lookup (name, (assoc, _, level)) =
      let fun lookup1 (name, scope) =
         case assoc (name, scope) of
            none => if scope = 0 then none else lookup1 (name, scope - 1) |
            mode => mode
      in
         lookup1 (name, level)
      end;

   allocArray (name, ub, env, typ) =
      let (assoc, newLoc, level) = env
      and id = (name, level) in
         case assoc id of
            none => ([id => array(newLoc, ub, typ)] assoc, newLoc + (ub * sizeOf (typ)), level) |
            _    => error env name "Variable already declared."
      end;

   allocEach (vars, env, typ) =
      let fun allocOne (declElt) =
                 let (assoc, newLoc, level) = env in
                    case declElt of
                       varDecl (name)       => allocVar (name, env, typ) |
                       arrayDecl (name, ub) => allocArray (name, ub, env, typ)
                 end
      in
         case vars of
            nil            => env |
            varElt :: rest => allocEach (rest, allocOne (varElt), typ)
      end;

     lvalue_error id msg = error (int_type, loadAddr (0)) id msg;
     exp_error exp msg   = error (int_type, loadInt (0)) exp msg;
     imp_error v msg     = error (null) v msg;
     bad_expr exp        = error (int_type, loadInt (0)) exp "Mismatched type in expression.";

    checkTypes (typ, (t1, v1), (t2, v2)) =
       if (typ = t1) andalso (typ = t2) then (v1, v2)
       else recover (loadInt (0), loadInt(0)) "Mismatched types in expression.";

semantic functions

     P : AST -> OUTPUTFILE;             (* program semantics *)
    D : AST -> ENV -> (ENV * DACTION);  (* declaration semantics *)
    V : AST -> DECL_ELEMENT list;        (* lists of vars in declarations *)
    B : AST -> ENV -> (ENV * IACTION);  (* scope block semantics *)
    C : AST -> ENV -> IACTION;           (* command semantics *)
    L : AST -> ENV -> (TYPE * LACTION); (* l-value expression semantics *)
    E : AST -> ENV -> (TYPE * VACTION); (* expression semantics *)

semantic equations

(* -- Programs *)
P [[ "program" name "(" inputfile "," outputfile ")" "is" body ]] =
   let ((_, memSize, _), programBody) = B [[body]] initEnv in
      execute (
        prog (memSize, sizeOf (int_type), sizeOf (bool_type), [[inputfile]], [[outputfile]], seq (programBody,
wrapup) ) )
   end;

(* -- Declarations *)
D [[ decl decls ]] env = let (env1, dact1) = D [[decl]] env in
                           let (env2, dact2) = D [[decls]] env1 in
                              (env2, declSeq (dact1, dact2))
                           end
                        end;

D [[ ]] env = (env, nullDecl);

D [[ "int" vars ]] env = (allocEach (V [[vars]], env, int_type), nullDecl);

D [[ "proc" name1 "(" name2 ")" "is" body ]] env =
   let (assoc, paramLoc, level) = env
   and id = (name1, level) in
      case assoc id of
         none => let procId    = makeToken name1
                 and nestedEnv = (assoc, paramLoc, level + 1)
                 and procEnv   = allocVar (name2, nestedEnv, int_type)
                 and (newEnv, bodyAct) = B [[body]] procEnv
                 and (newAssoc, newLoc, _) = newEnv
                 in
                   ( ([id => procedure (paramLoc, procId)] newAssoc, newLoc, level), proc (procId, bodyAct) )
                 end |
         _ => error (env, noProcAction) name1 "Procedure already declared."
   end;

(* -- lists of variables *)
V [[ varElt "," varElts ]] = V [[varElt]] :: V [[varElts]];

V [[ ]] = nil;

V [[ id ]] = varDecl [[id]];

V [[ id "[" num "]" ]] = arrayDecl ([[id]], [[num]]);

(* -- Blocks *)
B [[ decls "begin" stmts "end" ]] env = let (senv, dact) = B [[decls]] env in
                                            (senv, block (dact, C [[stmts]] senv))
                                         end;

(* -- Commands *)
C [[ stmt ";" stmts ]] env = seq (C [[stmt]] env, C [[stmts]] env);

C [[ ]] env = null;

C [[ lvar ":=" expr ]] env = let (ltype, lvalue) = L [[lvar]] env
                             and (rtype, rvalue) = E [[expr]] env in
                                if ltype = rtype then
                                   case ltype of
                                      int_type  => storeInt (lvalue, rvalue) |
                                      bool_type => storeBool (lvalue, rvalue)
                                else
                                   imp_error lvar "Type mismatch in assignment."
                             end;

C [[ "while" expr "do" stmts ]] env =
   let (typ, value) = E [[expr]] env in
```

```
      case typ of
         bool_type => while (value, C [[stmts]] env) |
         _          => imp_error expr "WHILE expression must be boolean."
    end;

C [[ "call" id "(" expr ")" ]] env =
   let (typ, value) = E [[expr]] env in
      case typ of
         int_type =>
            (case lookup ([[id]], env) of
               procedure (argLoc, name) => seq ( storeInt ((loadAddr argLoc), value), call (name) ) |
               _                         => imp_error id "Identifier not declared as a procedure.") |
         _ => imp_error expr "Procedure argument must be integer."
   end;

(* -- L-Values *)
L [[ id ]] env =
   case lookup ([[id]], env) of
      none          => lvalue_error id "Variable not declared." |
      var (loc, typ)  => (typ, loadAddr (loc)) |
      array (_)       => lvalue_error id "Missing an array subscript.";

(* -- Expressions *)
E [[ id ]] env =
   case lookup (id, env) of
      none          => exp_error id "Variable not declared." |
      var (loc, typ)  => (case typ of
                            int_type  => (typ, fetchInt (loc)) |
                            bool_type => (typ, fetchBool(loc)) ) |
      array (_)       => exp_error id "Missing an array subscript.";

E [[ id "[" expr "]" ]] env =
   let (typ, value) = E [[expr]] env in
      case typ of
         int_type => (case lookup (id, env) of
                        none    => exp_error id "Array not declared." |
                        var (_) => exp_error id "Variable not declared as an array." |
                        array (loc, ub, typ) =>
                           (case typ of
                              int_type  => (typ, contInt (indexInt (loc, check (ub, value)))) |
                              bool_type => (typ, contBool (indexBool (loc, check (ub, value)))))) |
         _          => imp_error expr "Array index must be integer."
   end;

E [[ expr1 "+" expr2 ]] env = let (v1, v2) = checkTypes (int_type, E [[expr1]] env, E [[expr2]] env) in
                                 (int_type, add (v1, v2))
                              end;

E [[ num ]] env = (int_type, loadInt (num));

end semantics
```

Appendix B: A Continuation-Style Microsemantics (excerpts)

```
microsemantics cont;

semantic domains

   (* Store locations are simply integers. *)
   LOC = INT;

   (* Array upperbounds are also integers. *)
   UPPERBOUND = INT;

   (* Integers and booleans are the only storable and expressible
    * values, and store locations may be uninitialized. *)
   SV     = union uninit | iValue of INT | bValue of BOOL;
   EV     = SV;
   MEMORY = LOC -> SV;

   (* The store consists of:
    *   - the memory mapping, input and output files
    *   - the maximum size of memory, in locations
   *   - the number of locations needed to store an integer, and a boolean *)
  STORE  = MEMORY * INPUTFILE * OUTPUTFILE * LOC * INT * INT;

  (* The dynamic environment is used for keeping track of
   * procedure body actions. *)
  DENV = TOKEN -> PROC_BODY;
  PROC_BODY = union noProc | procBody of IACTION;

  (* The continuations. *)
  ANSWERS = OUTPUTFILE;
  CONT    = STORE -> ANSWERS;  (* command continuations *)
  KONT    = EV -> CONT;        (* expression continuations *)
  LCONT   = LOC -> CONT;       (* l-value continuations *)
  DCONT   = DENV -> CONT;      (* declaration continuations *)

action domains

  IACTION = DENV -> CONT -> CONT;  (* imperative actions *)
  VACTION = KONT -> CONT;          (* value-producing actions *)
  LACTION = LCONT -> CONT;         (* location-producing actions *)
  DACTION = DCONT -> CONT;         (* declaration actions *)

auxiliary functions

  (* Perform the given arithmetic operation.
   *    ((INT * INT) -> INT) * VACTION * VACTION) -> VACTION *)
  infix oper;

  doIntOp (op oper, v1, v2) =
     fn k. v1 { fn e1. v2 { fn e2.
        let iValue (i1) = e1 and iValue (i2) = e2 in
           k (iValue (i1 oper i2))
        end } };

  nonfix oper;

  (* Retrieve the contents of the store at the given location.
   *    (LOC * STORE) -> EV *)
  contents (loc, s) =
     let (m, _, _, _, _, _) = s in
        case (m loc) of
           uninit     => fatal ("Referencing uninitialized location.", loc) |
           v          => v
     end;

  (* Update the store location with the given value.
   *    (LOC * SV * STORE) -> STORE *)
  update (loc, v, (m, inp, out, msz, isz, bsz)) = ([loc => v] m, inp, out, msz, isz, bsz);

  (* Given the location-producing and value-producing actions,
   * compute the location and value, and then store the value.
   *    (LACTION * VACTION) -> IACTION *)
  storeVal (l, v) = fn denv. fn c. l { fn loc. v { fn ev. { fn s. c (update (loc, ev, s)) } } };

  (* Load the contents of the given location.
   *    LOC -> VACTION *)
  fetchVal (loc) = fn k. { fn s. k (contents (loc, s)) s };

  (* Given a location-producing action, compute the location and
   * load its contents.
   *    LACTION -> IACTION *)
  loadContents (l) = fn k. l { fn loc. { fn s. k (contents (loc, s)) s } };

  (* Combine the two declaration environments into one. *)
  combine (denv1, denv2) tok = case denv1 tok of
                                  noProc => denv2 tok |
                                  p      => p;

  nullDenv : DENV = fn name. noProc;     (* The null declaration environment. *)

  nilStore = ():STORE;                   (* The "irrelevant" store. *)

  nilCont  = fn s. ():ANSWERS;           (* The "irrelevant" continuation. *)
```

```
   (* The required "execute" function.
    *     IACTION -> OUTPUTFILE *)
   execute (a) = a nullDenv nilCont nilStore;

operators

(* Basic arithmetic operators. *)
add : VACTION * VACTION -> VACTION is
add (v1, v2) = doIntOp (op +, v1, v2);

(* Load the integer/boolean constant. *)
loadInt : INT -> VACTION is
loadInt (n) = fn k. k (iValue n);

(* Load the contents of the location. *)
fetchInt : LOC -> VACTION is
fetchInt (loc) = fetchVal (loc);

(* Load the location (not its contents). *)
loadAddr : LOC -> LACTION is
loadAddr (loc) = fn l. { fn s. l loc s };

(* Compute the location and load its contents. *)
contInt : LACTION -> VACTION is
contInt (l) = loadContents (l);

(* Compute the location and value, and then store the value. *)
storeInt : (LACTION * VACTION) -> IACTION is
storeInt (l, v) = storeVal (l, v);

(* Perform each action in sequence. *)
seq : (IACTION * IACTION) -> IACTION is
seq (a1, a2) = fn denv. fn c. a1 denv { a2 denv c };

(* Loop until the test expression evaluates to false *)
while : (VACTION * IACTION) -> IACTION is
while (v, a) = fn denv. fn c. v { fn ev.
   let bValue (b) = ev in
      if b then a denv (while (v, a) denv c) else c
   end };

(* Perform the given imperative action after taking care of the
 * declaration actions.  I.e., open a new scope block. *)
block : (DACTION * IACTION) -> IACTION is
block (dact, act) = fn denv. fn c. dact { fn denv1. act (combine (denv, denv1)) c };

(* Call the procedure *)
call : TOKEN -> IACTION is
call (name) = fn denv. fn c. { case denv name of
                                  procBody (body) => body denv c |
                                                  => fatal ("Can't find procedure.", name) };

(* Declare the procedure *)
proc : (TOKEN * IACTION) -> DACTION is
proc (name, body) = fn dcont. dcont ([name => procBody (body)] nullDenv);

end microsemantics
```

Appendix C: A KleinPL Compiler (excerpts)

```
; Generated by MESS from macrosemantic specification file kleinpl.mes

(DEFINE (!E AST)
  (LET ((NODE (CAR AST))
        (ARGS (CDR AST)))
    (CASE NODE
      (NUM
       (APPLY
          (LAMBDA (!NUM)
            (LAMBDA (!ENV)
              (TUPLE !INT_TYPE (LIST '!LOADINT !NUM))))
          ARGS))
      (|EXPR1 "+" EXPR2|
        (APPLY
           (LAMBDA (!EXPR1 !EXPR2)
             (LAMBDA (!ENV)
               (LET* ((V (!CHECKTYPES (TUPLE !INT_TYPE ((!E !EXPR1) !ENV) ((!E !EXPR2) !ENV))))
                      (!V1 (CAR V))
                      (!V2 (CADR V)))
                 (TUPLE !INT_TYPE (LIST '!ADD (LIST 'TUPLE !V1 !V2))))))
           ARGS)))))

(DEFINE (!C AST)
  (LET ((NODE (CAR AST))
        (ARGS (CDR AST)))
    (CASE NODE
      (|"while" EXPR "do" STMTS|
       (APPLY
          (LAMBDA (!EXPR !STMTS)
            (LAMBDA (!ENV)
              (LET* ((V ((!E !EXPR) !ENV))
                     (!TYP (CAR V))
                     (!VALUE (CADR V)))
                ((LAMBDA (V)
                   (COND ((EQ? V !BOOL_TYPE) (LIST '!WHILE (LIST 'TUPLE !VALUE ((!C !STMTS) !ENV))))
                         (ELSE ((!IMP_ERROR !EXPR) "WHILE expression must be boolean."))))
                 !TYP))))
          ARGS))
      (|LVAR ":=" EXPR|
       (APPLY
          (LAMBDA (!LVAR !EXPR)
            (LAMBDA (!ENV)
              (LET* ((V ((!L !LVAR) !ENV))
                     (!LTYPE (CAR V))
                     (!LVALUE (CADR V))
                     (V ((!E !EXPR) !ENV))
                     (!RTYPE (CAR V))
                     (!RVALUE (CADR V)))
                (IF (!= !LTYPE !RTYPE)
                    ((LAMBDA (V)
                       (COND ((EQ? V !INT_TYPE) (LIST '!STOREINT (LIST 'TUPLE !LVALUE !RVALUE)))
                             (ELSE (LIST '!STOREBOOL (LIST 'TUPLE !LVALUE !RVALUE)))))
                     !LTYPE)
                    ((!IMP_ERROR !LVAR) "Type mismatch in assignment.")))))
          ARGS)))))
```

Appendix D: The Bubble-Sort Program and Object Code (excerpts)

```
program bubbleSort ( "stdin", "stdout" ) is
{* This KleinPL program sorts a sequence of integers.
 * Integers are read in until either a "-999" value, or 20 elements, are read. *}
int
   num [20];     {* The array to sort *}

   proc sort (numElts) is
   {* This procedure uses a bubble sort to arrange the 0..numElts-1 elements of the "num" array. *}

      proc switch (idx) is
      {* Exchange num[idx] with num[idx+1]. *}
      int temp;
      begin
         temp := num [idx];
         num [idx] := num [idx + 1];
         num [idx + 1] := temp
      end; {* switch *}

   int last, current;

   begin {* Main body of sort *}
      last := numElts - 1;
      while last >= 1 do
         current := 0;
         while current < last do
            if num [current] < num [current + 1] then
               call switch (current)
            end if;
            current := current + 1
         end while;
         last := last - 1
      end while
   end; {* sort *}

   proc printNums (size) is
   {* Dump out the array, 0 .. size-1. *}
   int i;
   begin
      i := 0;
      while i < size do
         write num [i];
         i := i + 1
      end while
   end; {* printNums *}

int numRead,      {* The number of integers read in *}
    val;          {* The current input *}

bool notDone;     {* Done reading? *}

{* Main program *}
begin
   numRead := 0;
   read val;
   notDone := val <> -999;

   while notDone do
      num [numRead] := val;
      numRead := numRead + 1;
      if numRead = 20 then
         notDone := false
      else
         read val;
         if val = -999 then
            notDone := false
         end if
      end if
   end while;

   call printNums (numRead);
   write -999;
   call sort (numRead);
   call printNums (numRead)
end
```

The following is the object code produced for the procedure "switch":

```
(!PROC
   (TUPLE
      '!SWITCH
      (!BLOCK
         (TUPLE
            (!DECLSEQ (TUPLE !NULLDECL !NULLDECL))
            (!SEQ (TUPLE
                     (!STOREINT (TUPLE (!LOADADDR 44)
                                       (!CONTINT (!INDEXINT (TUPLE 0 (!CHECK (TUPLE 20 (!FETCHINT 42)))))))
                     (!SEQ (TUPLE
                              (!STOREINT
                                 (TUPLE
                                    (!INDEXINT (TUPLE 0 (!CHECK (TUPLE 20 (!FETCHINT 42)))))
                                    (!CONTINT
                                       (!INDEXINT
                                          (TUPLE
                                             0
                                             (!CHECK
                                                (TUPLE 20 (!ADD (TUPLE (!FETCHINT 42) (!LOADINT 1))))))))))
                              (!SEQ (TUPLE
                                       (!STOREINT
                                          (TUPLE
                                             (!INDEXINT
                                                (TUPLE
                                                   0
                                                   (!CHECK
                                                      (TUPLE
                                                         20
                                                         (!ADD (TUPLE (!FETCHINT 42) (!LOADINT 1))))))
                                             (!FETCHINT 44)))
```

Appendix E: The MESS-Generated Implementation of the Microsemantics

```
; Generated by MESS from microsemantic specification file cont.mes

(DEFINE !ADD
  (REC !ADD
    (LAMBDA (|v7|)
      (LET ((!V1 (CAR |v7|))
            (!V2 (CADR |v7|)))
        (!DOINTOP (TUPLE !+ !V1 !V2))))))

(DEFINE !STOREINT
  (REC !STOREINT
    (LAMBDA (|v20|)
      (LET ((!L (CAR |v20|))
            (!V (CADR |v20|)))
        (!STOREVAL (TUPLE !L !V))))))

(DEFINE !SEQ
  (REC !SEQ
    (LAMBDA (|v22|)
      (LET ((!A1 (CAR |v22|))
            (!A2 (CADR |v22|)))
        (LAMBDA (!DENV)
          (LAMBDA (!C)
            ((!A1 !DENV) ((!A2 !DENV) !C))))))))

(DEFINE !WHILE
  (REC !WHILE
    (LAMBDA (|v24|)
      (LET ((!V (CAR |v24|))
            (!A (CADR |v24|)))
        (LAMBDA (!DENV)
          (LAMBDA (!C)
            (!V (LAMBDA (!EV)
                  (LET* ((V !EV)
                         (!B (CDR V)))
                    (IF !B
                        ((!A !DENV) (((!WHILE (TUPLE !V !A)) !DENV) !C))
                        !C))))))))))

(DEFINE !CALL
  (REC !CALL
    (LAMBDA (!NAME)
      (LAMBDA (!DENV)
        (LAMBDA (!C)
          ((LAMBDA (V)
             (COND ((EQ? '!!!PROCBODY (CAR V))
                    (LET ((!BODY (CDR V)))
                      ((!BODY !DENV) !C)))
                   (ELSE (!FATAL (TUPLE "Can't find procedure." !NAME)))))
           (!DENV !NAME)))))))

(DEFINE !PROC
  (REC !PROC
    (LAMBDA (|v27|)
      (LET ((!NAME (CAR |v27|))
            (!BODY (CADR |v27|)))
        (LAMBDA (!DCONT)
          (!DCONT
             (LAMBDA (V)
               (IF (EQUAL? V !NAME) (!PROCBODY !BODY) (!NULLDENV V)))))))))
```