A Realistic Compiler Generator Based on High-Level Semantics

Another Progress Report

Peter Lee
The University of Michigan
Peter_Lee%UMich-MTS.mailnet@MIT-multics.ARPA
pete@CAEN.engin.UMich.EDU

Uwe Pleban
PhiloSoft *and* The University of Michigan
Uwe_Pleban%UMich-MTS.mailnet@MIT-multics.ARPA

Department of Electrical Engineering and Computer Science
Division of Computer Science and Engineering
The University of Michigan
Ann Arbor, Michigan 48109-2122

Received 10/31/86

Abstract

We have developed a new style of semantic definition called *high-level semantics*. In constrast to traditional denotational semantics, high-level semantics is suitable for both *defining* the functional *meaning* of programming languages, as well as *describing* realistic compiler *implementations*. Moreover, high-level specifications are considerably more descriptive and intelligible than traditional specifications.

This paper describes the compiler generator MESS, which embodies the principles of high-level semantics. MESS has been used to generate compilers for nontrivial languages. The compilers are efficient, and produce object programs that are competitive with those generated by hand-written compilers.

1 Overview

A number of recently developed semantics directed compiler generators [Mos79,Pau82,Wan84] are based on denotational semantics [MiS76,Sto77]. Experience with these systems shows that the generated compilers, and the object programs they produce, are several orders of magnitude less efficient than their hand-written counterparts. We have discovered that this inefficiency is an *inherent* property of the compilers, due to several fundamental problems with the specification techniques used in traditional denotational semantics. Furthermore, we find that many of these problems make it difficult to write, modify, and debug the semantic specifications.

In order to solve these problems, we have developed a new style of semantic definition called *high-level semantics*. In constrast to traditional denotational semantics, high-level semantics is suitable for both *defining* the functional *meaning* of programming languages, as well as *describing* realistic compiler *implementations*.

To verify the feasibility of our ideas, we have implemented a semantics directed compiler generator called MESS which automatically derives compiler implementations from high-level semantic descriptions of programming languages. The generated compilers are *realistic* because they have the following properties:

- They compile nontrivial sequential programming languages into object code for standard machine architectures.
- Their internal structure is similar to that exhibited by hand-crafted compilers: They have a multipass structure, and perform the usual compile time computations, such as type checking, during compilation.
- Both the compilers and the object programs they produce exhibit good performance characteristics. In particular, the size and speed of the object programs are competitive with those produced by hand-crafted compilers.

The structure of this paper is as follows. The next section provides a brief overview of high-level semantics. Section 3 describes the MESS system, which embodies the fundamental principles of high-level semantics. It also compares the performance of MESS with that of another semantics-based compiler generator, and the performance of a MESS-generated compiler with that of commercially available compilers. A detailed example of a MESS specification and how a compiler is generated from it is the subject of Section 4. In Section 5, we discuss extensions to MESS and other topics of current research. Finally, we summarize related work in Section 6.

2 Overview of High-Level Semantics

A complete exposition of high-level semantics appears in [Lee86]. Here, we provide just a brief overview. We start with a critique of the traditional denotational approach, and then list the salient characteristics of the high-level approach.

2.1 Critique of Traditional Denotational Semantics

We assume familiarity with traditional denotational semantics, as explicated in [MiS76], [Sto77], and [Gor79]. We say *traditional* denotational semantics because we are referring not only to the mathematical basis of the approach, but also the notation, writing style, and "tricks of the trade" used in denotational specifications.

Our criticism of this approach can be summarized as follows:

1. The writing style inextricably intertwines all the details of the semantic model (such as the structure of environments and stores, the use of direct or continuation style models, etc.) with the actual semantics of the programming language. This usually necessitates a complete reformulation of the semantic specification for a language when features are added whose description requires a more powerful semantic model.[1] In addition, certain aspects of the semantics are needlessly overspecified, such as the handling of storage allocation. By not separating model-dependent details from the actual semantics, the extensibility of a given semantic definition is severely compromised. We call this problem *lack of separability.*
2. Traditional specifications are not formulated in terms of fundamental concepts expressing the design, analysis, use, and implementation of programming languages. Instead, these concepts are always encoded via function abstraction and application. This results in convoluted functionalities, excessive use of anonymous λ-abstractions, and various "currying tricks" which make it extremely difficult to write, read, and debug semantic equations. Moreover, the distinction between static (compile time) and dynamic (runtime) language components is blurred, as both are expressed within the same framework. Clearly, this constitutes *poor semantics engineering.*
3. In a similar vein, the semantic aspects of certain language constructs, such as environment manipulation at routine entry and exit, the distinction between variables and parameters, and others, are either completely ignored or expressed in highly cryptic terms. We call this *minimalistic semantic explication.*
4. Existing denotational definitions are largely monolithic without any provisions for expressing a language specification as a collection of semantic modules. This is *lack of modularity.*

There are a number of important consequences of these deficiencies for the automatic generation of prototype implementations from traditional denotational specifications.

1. Expressing standard denotational descriptions in Scott's language Lambda essentially amounts to writing in a very primitive assembly language (with more or less syntactic maple syrup) for a λ-calculus machine, with all the accompanying drawbacks of assembly language coding. The *lack of separability* forces one to implement a traditional semantics by *emulating* a λ-calculus machine, both at compile time and at runtime. This emulation is done by means of a partial evaluator performing β-reductions. Not surprisingly, this is the approach taken by all implementation generators based on denotational semantics, including SIS [Mos79], PSP [Pau81,Pau82], SPS [Wan84], and Watt's use of ML as a semantic metalanguage [Wat84].
2. The *poor semantics engineering* has two major negative effects. First, the emulation of the details of the semantic model requires an excessive number of function closures, resulting in *extremely slow execution* of the "target code" derived from the semantics of a program. Observations by Paulson [Pau81], Wand [Wan84], and Watt [Wat84], and our own extensive experiments with SIS [BoB82] and PSP [Ple84a] indicate that such emulation of Lambda assembly code always runs between 500 and several thousand times slower than equivalent machine code which is directly executable. Second, the blurring of static and dynamic language concepts requires the presence of a partial evaluation mechanism in a compiler generated from a traditional semantics. Consequently, the generated compilers run considerably slower than handwritten non-optimizing compilers. [LeP86b]
3. The *minimalistic semantic explication* makes it *impossible to mechanically derive efficient implementations* which exploit the richness of existing machine architectures. Curiously enough, the same problem is caused by some of the overspecification of storage models in the traditional denotational approach.
4. The lack of modularity makes it very difficult to engineer semantic specifications in the sense of writing, testing, and debugging them. For realistic languages, semantic specifications may well be several thousand lines long, and hence their development benefits greatly from applying to them standard software engineering principles such as modularization, information hiding, and data abstraction.

Several of our criticisms of traditional denotational semantics have already been voiced by other researchers.

[1] A striking example of this is given in Chapters 9 through 11 in Stoy's book, where language extension forces such rewriting five times.

For example, Mosses' abstract semantic algebra approach to semantics [Mos82,Mos84] is a direct outgrowth of his dissatisfaction with the intertwining of model dependent details and the actual semantics in denotational specifications. Watt [Wat82] has studied modularity issues in semantic specifications. To our knowledge, however, no comprehensive analysis of the negative impact of traditional denotational definitions on the generation of compiler implementations has ever appeared in print.

2.2 Characteristics of High Level Semantics

We now summarize the salient characteristics of our high-level approach to denotational semantics, which remedies all of the deficiencies of the traditional approach.

1. *Separability.* The semantic definition of a programming language is cleanly separated from the details of underlying semantic models. This separation is enforced by requiring two distinct specifications, called *macrosemantics* and *microsemantics.*
2. *Description via action-based operators.* The macrosemantics (semantics, for short) of a language is defined by homomorphically combining the denotations of syntactic constructs with action-based semantic operators to form the semantic equations. These operators yield declarative, imperative, or value producing actions, akin to those discussed in [ClF82,Mos82,Mos84].
3. *Information hiding.* Suitable microsemantic definitions provide the interpretation for the operators. However, only the names and signatures of the operators are made available to the macrosemantics. Thus, the principle of separability guarantees the *invariance* of the macrosemantics under different microsemantic specifications of the operators.
4. *Distinction between compile time and runtime.* The separation of the semantics into macrosemantics and microsemantics automatically distinguishes compile time objects from runtime objects. In essence, all domains defined in the macrosemantics are compile time domains, while those defined in the microsemantics are runtime domains. Thus, we obtain the same effect as in the two-level metalanguage TML. [NiN86]
5. *Modularity.* In addition to the kind of modularity provided by the separation of the microsemantic details from the semantics, high-level semantic descriptions may exhibit two other kinds of modularity. First, the static semantics may be separated from the dynamic semantics. Second, any (macro or micro) semantic specification may be written as a collection of semantic modules, whose interfaces are subject to consistency checking. For example, there may be a module for the semantics of imperative constructs, one for expressions, and one for declarations. This allows for the incremental development of semantic definitions. As an additional benefit, microsemantic modules may be reused as "semantic libraries."
6. *Exposition of implementation structures.* The semantic operators are chosen to reflect not only fundamental language concepts, but also fundamental implementation concepts. Consequently, the operators can be efficiently implemented, e.g., by interpreting them as templates of intermediate code for a code generator. In addition, the distinction between the static and dynamic components of a language obviates the need for partial evaluation by means of β-reduction during compilation. [LeP86b]
7. *Extensibility.* Language features requiring new operators can be readily accommodated by extending the microsemantic modules. This may require a rewriting of (parts of) the microsemantics, but always leaves the existing macrosemantics intact. The portions requiring rewriting are easily identified.
8. *Simplified congruence proofs.* Demonstrating the congruence of two different semantic specifications merely involves relating the microsemantic specifications. The techniques described in [Roy86] should prove particularly appropriate in this context.
9. *Readability.* Finally, our specifications are written in a readable notation based on ML [Mil85], which allows them to be processed by the MESS system.

3 The MESS System

A compiler can be generated from a high-level semantics by treating the macrosemantics as a specification of the translation component of a compiler, and the microsemantics as a specification of an abstract machine or runtime environment. Figure 1 gives a pictorial depiction of this process as implemented in the MESS system. The figure shows that a macrosemantics specifies a translation from source abstract syntax trees (ASTs) to *prefix-form operator expressions* (POEs). These expressions can either be viewed as an algebraic specification of elements of a data type, or as unevaluated function applications. An interpretation for the operators is provided by a microsemantic definition.

With respect to realistic compilation, the macrosemantics specifies all of the compile time computations involved with translating abstract syntax trees to intermediate code. A suitable microsemantics, then, specifies further translation to target code.

The principle of separability dictates that *only* the names and signatures of the operators be shared between a macrosemantics and a microsemantics. Thus, any two microsemantic specifications that define the same operator names and signatures are interchangeable. This is depicted in Figure 1, where four different, yet compatible, microsemantic specifications are shown. A microsemantics might be based on standard denotational models, e.g., by writing λ-calculus definitions of the operators in the traditional direct or continuation styles. In this case the composition of the macro and micro semantics constitutes a standard denotational semantics. On

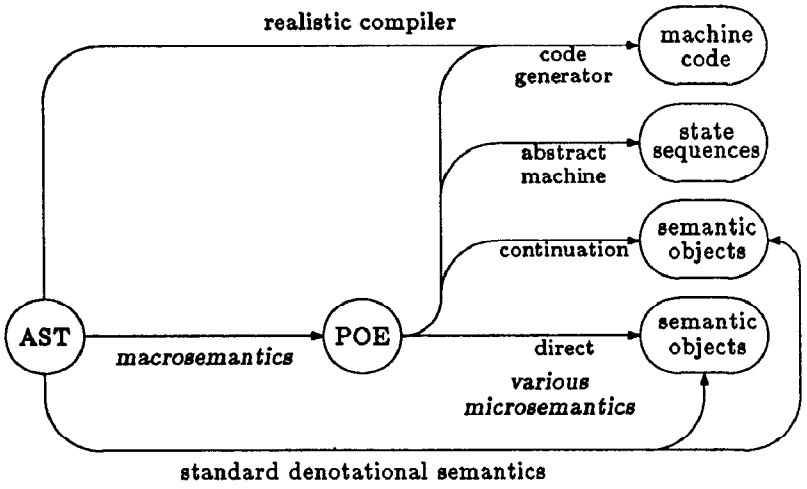


Figure 1: Compiler generation from high-level semantics

the other hand, one may choose an "operational" specification of the microsemantics by providing an abstract machine which is capable of interpreting the POEs. Finally, since the operator expressions are in prefix form, they can be processed by a code generator in a straightforward way.

3.1 MESS at Work

Figure 2 shows the overall structure of MESS, which runs on an IBM Personal Computer. The generated compiler front-ends and the front-end generator itself are written in Pascal. The semantics analyzer is written in SCHEME [StS78], as are the compiler cores it generates. We use the Turbo Pascal compiler [Bor85] and TI PC SCHEME system [TI85]. For the time being we write code generators in Prolog, using the Turbo Prolog implementation [Bor86].

As a nontrivial test of our ideas, we have written a high-level semantics for HypoPL, a hypothetical language with arrays, nonrecursive procedures, and the usual Pascal-like control structures. In addition to the macrosemantic specification for HypoPL, we have written four different microsemantic specifications, corresponding to the four microsemantic specifications depicted in Figure 1: (1) a λ-calculus based direct style specification, (2) a λ-calculus based continuation style specification, (3) an abstract machine written in SCHEME, and (4) a code generator for the iAPX8086, the CPU used in the IBM PC.

We have ported Paulson's Semantics Processor (PSP) [Pau82,Pau81] to the IBM PC in order to compare its performance with that of MESS. PSP was chosen because it exhibits, to our knowledge, the best performance characteristics of any compiler generator based on denotational semantics. [Ple84a]

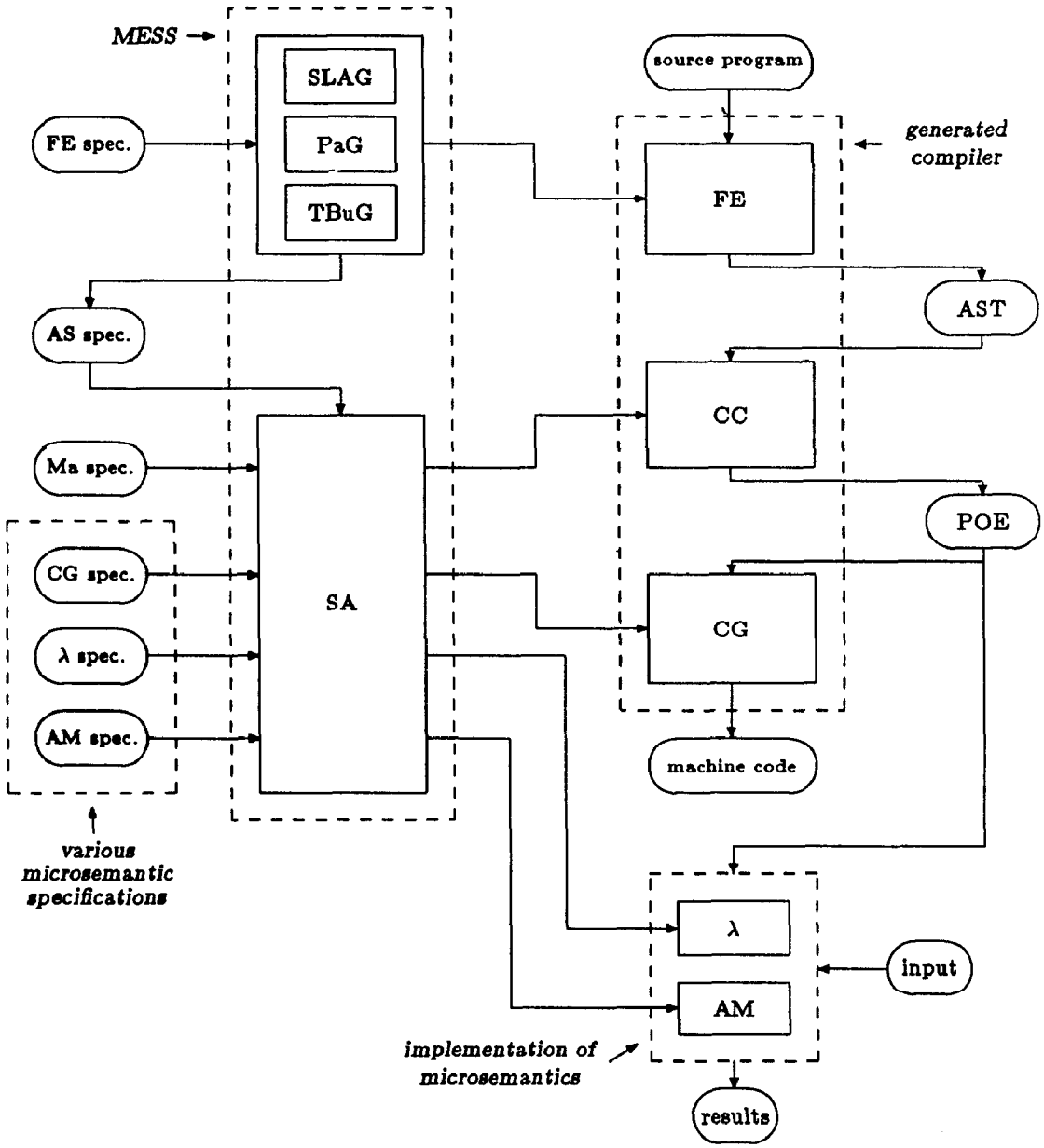
The time required to generate a HypoPL compiler by each system is given Figure 3. In this paper, all timings are given in seconds, and were recorded on an IBM PC with a 10MHz 8086 processor, 512 kilobytes of random-access memory, and a 10 megabyte hard disk with an access time of 65 milliseconds. The semantic specification used for the PSP timing is written in the continuation style, as is the microsemantic specification used for the MESS timing. Combined, the macro and micro semantic specifications processed by MESS are about 900 (well commented) lines long, and required approximately 24 man-hours to write and debug. Much of this time was spent locating type errors since at the time the MESS type checker was not complete. The PSP specification is about 1100 lines long.

As Figure 3 shows, MESS takes considerably longer to generate a HypoPL compiler than does PSP. This is due in large part to the fact that PSP is written in Pascal and was thus compiled to native machine code, whereas MESS is written in an implementation of SCHEME that compiles to interpreted byte codes. We expect that a native code compiler for SCHEME would allow MESS to run at a speed comparable to that of PSP.

Figure 4 shows compile times for an 82 line HypoPL bubble sorting program. Here we can see that the MESS-generated compiler is more than five times faster than the PSP-generated compiler.[2] Of its 105 seconds of compile time, the PSP-generated compiler spends more than 66 seconds performing β-reductions. In addition, it generates more than 6,000 instructions for the SECD machine [Lan64].

In order to make a comparison with hand crafted compilers, we have written equivalent bubble sorting programs in C, Pascal, and Modula-2, and then measured

[2]For both the MESS-generated and PSP-generated compiler timings, unnecessary I/O overhead time has been discounted.



This pictorial representation of MESS shows the various phases of compiler generation. The semanticist provides specifications for the front-end, the macrosemantics, and the microsemantics (FE spec., Ma spec., and one of CG spec., λ spec., or AM spec.). A specification of the abstract syntax (AS spec.) may also be given, although the front-end generator, consisting of the Simple Lexical Analyzer Generator, Parser Generator, and Tree-Builder Generator (SLAG, PaG, and TBuG) generates this automatically.

The semantics analyzer (SA) processes the semantic descriptions and generates the compiler core (CC). In addition, it produces an implementation of the microsemantic operators in a form corresponding to the type of microsemantics specified: either a code generator (CG), a set of functions written in the λ-calculus (λ), or an abstract machine (AM).

If a code generator is produced, the combination of FE+CC+CG constitutes a realistic compiler. Otherwise, FE+CC is a compiler that produces prefix-form operator expressions (POEs) which must be interpreted by either a λ-calculus machine enriched with the λ-functions, or else the abstract machine (AM).

Figure 2: Our big MESS.

Figure 3: Compiler generation times for HypoPL (in seconds)

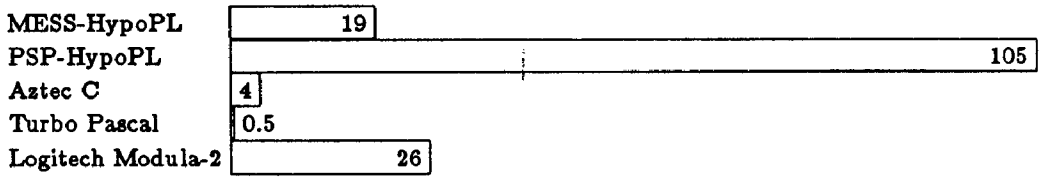


Figure 4: Compile times for bubble sort program (in seconds)

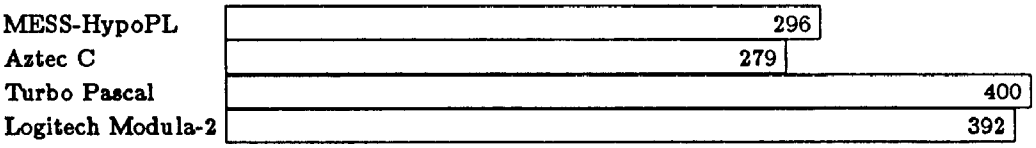


Figure 5: Object code sizes for bubble sort program (in bytes)

(all checking disabled)

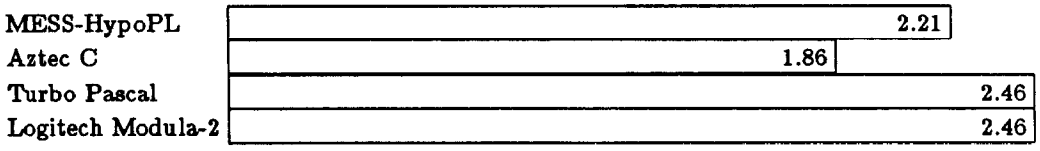


Figure 6: iAPX8086 object code execution times for sorting 250 integers (in seconds)

(all checking disabled)

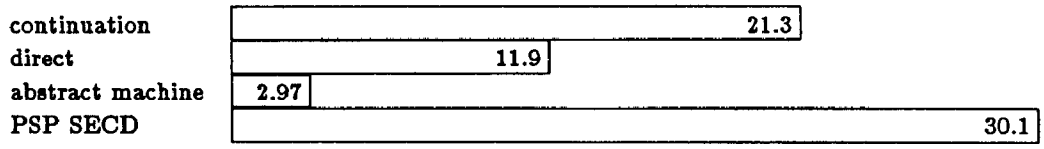


Figure 7: POE and SECD execution times for sorting 10 integers (in seconds)

the compile times for commercially available compilers for these languages. We use the following compilers: Manx Aztec C, Version 3.20e [Man85], Borland Turbo Pascal, Version 3.0 [Bor85], and Logitech Modula-2, Version 2.0 [Log86]. The timing for the Manx compiler measures only the compilation to symbolic assembly code. The Logitech compiler used is the fully-linked version, M2C. As Figure 4 shows, the MESS-generated HypoPL compiler is about five times slower than the C compiler, and faster than the Modula-2 compiler. The separate passes of MESS-generated compilers currently communicate via disk files since there is no provision for direct data communication between Pascal, SCHEME, and Prolog programs. This slows down the compilation considerably, perhaps by more than a factor of two. We look forward to the day when all components of the generated compilers are written in the same LISP dialect.

The MESS timing includes the time required to generate machine code for the iAPX8086 processor. Thus, we can compare the iAPX8086 object programs produced by the MESS-generated, C, Pascal, and Modula-2 compilers. This is done in Figures 5 and 6. The object codes sizes, in bytes, are given in Figure 5. We are pleased to point out that the object code produced by the MESS-generated compiler is more compact than that produced by the Turbo Pascal and Modula-2 compilers. The execution times for the bubble sort object programs produced by the various compilers are displayed in Figure 6. This graph shows the time required by each program to sort a worst case input of 250 integers. Again, we would like to point out that the performance of the object code produced by the MESS-generated compiler compares favorably with that produced by the hand-crafted compilers.

Due to memory constraints in the IBM PC, we are unable to run the PSP-generated SECD object program on an input of 250 integers. Also, we are unable to use the continuation and direct style microsemantics to interpret the POE produced by the MESS-generated compiler for such a large input. We believe this to be caused by a bug in the SCHEME system concerning a stack segment overflow during garbage collection. However, on a worst case input of ten integers, we have the timings given in Figure 7. Note that both of the MESS-generated microsemantic implementations based on standard λ-calculus models ("direct" and "continuation") exhibit better runtime performance than the SECD machine used by PSP.

4 An Example

In order to better illustrate the process of compiler generation in the MESS system,[3] we take as an example a tiny fragment of a high-level semantic specification. The fragment describes an imperative language with integers and one-dimensional arrays (and no other types). Although this language is not very realistic, it suits our illustrative purposes. Complete examples of large specifications can be found in [Lee86].

[3]Incidentally, the name, MESS, comes from the fact that high-level semantic specifications are *M*odular, *E*xtensible, and *S*eparated *S*emantic specifications.

4.1 The Macrosemantics

Macrosemantic specifications begin with a description of the compile time semantic domains. Of particular interest here is the definition of the static environment, S_ENV, which may be viewed as the central compile time data structure.

```
semantic domains

   S_ENV      = IDENT -> MODE;
   MODE       = union
      noneM |
      scalarM of NAME |
      arrayM  of (NAME * UPPERBOUND);
   UPPERBOUND = INT;
```

The domains IDENT and NAME are predefined and represent, respectively, the domains of syntactic identifiers and semantic names. The principle of separability dictates that syntactic identifiers be converted into semantic names before being passed on to any microsemantic operators. The nature of this conversion directly reflects the scoping properties of the language being described. [Lee86,PlL86] For example, in a lexically-scoped language, identifiers are α-converted to semantic names.

```
semantic functions

   P : AST -> OUTPUTFILE;
   D : AST -> S_ENV -> (S_ENV * DACTION);
   B : AST -> S_ENV -> IACTION;
   C : AST -> S_ENV -> IACTION;
   L : AST -> S_ENV -> LACTION;
   E : AST -> S_ENV -> VACTION;
```

The semantic functions section provides declarations for the functionalities of the semantic valuation functions: P (program meanings), D (declarations), B (blocks), C (statements), L (L-values), and E (expressions). Here we see references to the action domains, DACTION (declaration actions), IACTION (imperative actions), LACTION (L-value producing actions), and VACTION (value producing actions). These action domains must be defined by the microsemantics.

Now, a few example semantic equations.

```
C [[ stmt ";" stmts ]] s_env =
   CmdSeq (C [[ stmt ]] s_env,
           C [[ stmts ]] s_env);

C [[ lvar ":=" expr ]] s_env =
   Assign (L [[lvar]] s_env,
           E [[expr]] s_env);

L [[ id "(" expr ")" ]] s_env =
   case s_env [[id]] in
      noneM =>
         error "variable not declared" |
      scalarM (_) =>
         error "missing array subscript" |
      arrayM (name, ub) =>
```

```
        Index (Array (name),
               Check (ub,
                      E [[expr]] s_env));

E [[ num ]] s_env = Const num;
```

These equations describe, in order, the semantics of statement sequencing, assignments, L-value array indexing, and integer constant expressions. The operators CmdSeq, Assign, Index, Array, Check, and Const are defined in the microsemantics.

Several points should be noted about the macrosemantics:

- Anonymous λ-abstractions never appear in the semantic equations. This means that a compiler derived from these equations need not deal with higher order functions.
- Storage allocation details, for instance the domain of store locations, are absent from the equations. This is quite unlike a traditional denotational specification. However, we believe it to be appropriate here since stores are not conceptually necessary to describe the semantics of such a simple language (and, actually, much more complicated languages as well). For compiler generation, this means that the generated compiler isn't "tied down" to any particular model of the store, or method of storage allocation.
- In fact, all details of the semantic model are absent from the macrosemantics. For instance, it is not apparent whether continuations are used to model flow of control. This information is hidden in the microsemantics, and thus the macrosemantics is protected from massive rewriting if the semantic model changes.
- Microsemantic operators are always applied to runtime domains such as integers, or else composed with other operators. They are never applied to macrosemantic domains, as this would violate the separability property. Such violations are not allowed by the MESS system.

4.2 A Continuation-Based Microsemantics

We now give excerpts of one possible microsemantic specification which supports the macrosemantics of the previous section. The specification is written using continuations.

```
semantic domains

   (* dynamic environments *)
   D_ENV       = NAME -> DENOTATION;

   (* denotations *)
   DENOTATION = union
      unbound |
      intValue of INT |
      arrayValue of INT -> NAME;

   (* expressible values *)
   EV          = INT;

   (* program answers *)
   ANSWER      = OUTPUTFILE;

   (* command, L-value and
      expression continuations *)
   CONT        = D_ENV -> ANSWER;
   LCONT       = NAME -> CONT;
   ECONT       = EV -> CONT;
```

These domain definitions define dynamic environments and the various types of continuations. Note that D_ENV takes the place of the more traditional "store." NAMEs may denote either integer or array values, and are also used as L-values. Array values are mappings of integers to the names (i.e., L-values) of the array elements.

```
action domains

   (* imperative actions *)
   IACTION  = CONT -> CONT;

   (* L-value producing actions *)
   LACTION  = LCONT -> CONT;

   (* value producing actions *)
   VACTION  = ECONT -> CONT;
```

The action domains section defines the structure of the action domains. MESS enforces the requirement that every microsemantic operator produce a value in one of these domains.

```
operators

   (* Perform actions in sequence *)
   CmdSeq : IACTION * IACTION -> IACTION is
   CmdSeq (a1, a2) =
      fn cont. a1 { a2 cont };

   (* Assign the value to the L-value *)
   Assign : LACTION * VACTION -> IACTION is
   Assign (l, v) =
      fn cont. fn d_env.
         l { fn name. v { fn ev.
            cont ([name =>
                   intValue (v)] d_env) } };

   (* Produce a new L-value by indexing the
      given l-value *)
   Index : LACTION * VACTION -> LACTION is
   Index (l, v) = fn lcont. fn d_env.
      l { fn name. v { fn ev.
         let arrayValue (array) = d_env name
         in
            lcont (array ev) d_env
         end } };

   (* Return the L-value of the array *)
```

```
Array : NAME -> LACTION is
Array (name) = fn lcont. lcont name;

(* Check that the given integer is
   greater than the value *)
Check : INT * VACTION -> VACTION is
Check (n, v) = fn econt. v { fn ev.
   if (ev < n) andalso (ev >= 0)
   then econt ev
   else error "subscript out of range" };

(* Convert the integer to an action *)
Const : INT -> VACTION is
Const (n) = fn econt. econt n;
```

Although complete definitions for the operators are given, *only the names and functionalities are used by MESS for processing the macrosemantics.*

4.3 Generating the Compiler

Note: For the purposes of illustration, we have taken the liberty of simplifying things a bit throughout this subsection. In particular, the generated compiler core is more complicated than shown in order to allow for static semantic error handling of high quality. This also has the effect of expanding the structure of abstract syntax trees.

Once a macrosemantics and a microsemantics have been written, a compiler is generated as follows. First, the microsemantics is processed by MESS. This results in a *microsemantics interface file* which contains the names and signatures of the operators. In addition, an implementation of the microsemantics is generated.

Then, the macrosemantics, together with the microsemantics interface file, is processed by MESS. This results in a compiler core, written in SCHEME. An excerpt of the compiler core generated from the macrosemantics given in Section 4.1 is shown in Figure 8. Notice that the names of the microsemantic operators are quoted — this prevents the evaluation of the runtime operators at compile time. With the runtime operators safely quoted, the macrosemantics can be completely evaluated without the need for an expensive partial evaluation mechanism such as β-reduction [LeP86b].

Hence, compilation of the following source statement:

```
x(2) := 5;
```

given the declaration:

```
int x(10);
```

proceeds as follows. First, it is translated into an abstract syntax tree by the front-end. The trees are written as SCHEME s-expressions:

```
(|lvar ":=" expr|
  (|id "(" expr ")"| (|id| x) (|num| 2))
  (|num| 5))
```

This is then translated by the compiler core (in particular, the function C) of Figure 8 into the POE:

```
(assign
  (index (array 'x#) (check 10 (const 2)))
  (const 5))
```

where we assume x# is the semantic name for the syntactic identifier x.

4.4 Executing the Compiled Programs

This POE can now be "executed" in a number of ways. First, MESS automatically generates an implementation of the operators from the microsemantics. For example, the definitions of the CmdSeq and Check operators given in Section 4.2 are translated by MESS into the following SCHEME code:

```
(define (CmdSeq a1 a2)
  (lambda (cont)
    (a1 (a2 cont))))

(define (Check n v)
  (lambda (econt)
    (v (lambda (ev)
         (if (and (< ev n) (>= ev 0))
             (econt ev)
             (error
               "subscript out of range"))))))
```

Alternatively, one can write an abstract machine definition by hand which interprets the POEs. For example, the CmdSeq and Check operators might be implemented by the following hand-written SCHEME code:

```
(macro CmdSeq
  (lambda (e)
    `(let ((s1 (lambda () ,(cadr e)))
           (s2 (lambda () ,(caddr e))))
       (begin (s1) (s2)))))

(define (Check n v)
  (if (and (< v n) (>= v 0))
      v
      (error "subscript out of range")))
```

Here, we have avoided the use of continuations, as they are presumably not necessary to implement the operators. Note that a macro is used to define CmdSeq since the order of evaluation of arguments in SCHEME is not defined.

Finally, since the POEs are prefix form expressions comprised solely of applications of operators, a code generator can be used to obtain machine code. We have written such a code generator, and it translates the POE given above into the following iAPX8086 machine code (assume that the base address of the array x is 502, and subscript checking is suppressed):

```
mov   ax,4
add   ax,OFFSET mem_[502]
mov   bx,ax
mov   WORD [bx],5
```

```
(define (C ast)
  (let ((nodename (car ast))
        (subnodes (cdr ast)))
    (case nodename

      (|stmt ";" stmts|
       (apply
         (lambda (stmt stmts)
           (lambda (s_env)
             (list 'CmdSeq ((C stmt) s_env) ((C stmts) s_env))))
         subnodes))

      (|lvar ":=" expr|
       (apply
         (lambda (lvar expr)
           (lambda (s_env)
             (list 'Assign ((L lvar) s_env) ((E expr) s_env))))
         subnodes))

      ... )))

(define (L ast)
  (let ((nodename (car ast))
        (subnodes (cdr ast)))
    (case nodename

      (|id "(" expr ")"|
       (apply
         (lambda (id expr)
           (lambda (s_env)
             ((lambda (v)
                (cond ((eq? v noneM) (error "variable not declared"))
                      ((eq? 'scalarM (car v)) (error "missing array subscript"))
                      (else (let ((name (cadr v))
                                  (ub (caddr v)))
                              (list 'Index
                                    (list 'Array name)
                                    (list 'Check ub ((E expr) s_env))))))))
              (s_env (cadr id)))))
         subnodes))

      ... )))

(define (E ast)
  (let ((nodename (car ast))
        (subnodes (cdr ast)))
    (case nodename

      (|num|
       (apply
         (lambda (num)
           (lambda (s_env)
             (list 'Const num)))
         subnodes))

      ... )))
```

Figure 8: Excerpt of generated compiler core

5 Unfinished Business

The following tasks are currently under investigation:

- Specification of code generators in the MESS metalanguage. MESS is currently being extended to be able to process microsemantic specifications which treat the microsemantic operators as data type constructors rather than higher-order functions. This will forego the need for using another language such as Turbo Prolog for specifying code generators. In addition, it will allow the generated compiler core and code generator for a language to communicate via data structures in memory rather than on disk files.
- Generation of compilers for larger languages. We are currently in the midst of generating a compiler for Sol/C (sort of like C), which features, among other things, recursive procedures with any number of reference or value parameters, integer, Boolean, and character objects, and multidimensional open array parameters. Although the macrosemantics for Sol/C and the code generator specification are complete, the current revisions to MESS have prevented us from including a complete set of experimental results in this paper. We plan to report on Sol/C in the near future.
- Specification of multi-pass compiler cores. MESS-generated compilers currently have a fixed, three-pass structure. Often, it would be convenient to specify more passes for the compiler core, for instance, to separate type checking concerns from the rest of the macrosemantics.
- Development of a microsemantic library. Such a library would define operator sets powerful enough to describe the semantics of most sequential programming languages. Essentially, this is a semantics-based approach to the UNCOL problem.

6 Related Work

Our work started after experimenting with the direct implementation of denotational specifications using SIS [BoB82] and PSP [Ple84a]. A first step towards high-level semantics was the development of normal form semantics [Ple84b]. This was directly inspired by Wand's work on deriving postfix code from continuation semantics [Wan82], and research in the area of code generator specifications languages [GlG78,Gan80,Bir82]. The connection with Mosses' concept of action-based semantic operators [Mos82,Mos84], although known for quite some time, was made only recently. Indeed, the continuation transformers constructed in [Ple84b] are directly analogous to semantic operators yielding actions.

There are two semantics-based compiler generators similar in spirit to our approach. The CERES system of Jones and Christiansen [JoC82] accepts semantic specifications expressed in a small number of action-oriented operators inspired by those of Mosses. Sethi's system [Set81] generates efficient compilers by treating fundamental "runtime" operators in the semantic specification as uninterpreted symbols. His work is also motivated by that of Mosses, but still refers to microsemantic concepts such as continuations and stores. Both systems have only been used for generating compilers for languages with control structures for sequencing, looping and decision making, and simple expressions. Also, the intermediate code produced by the generated compilers must be translated by a code generator in an ad hoc manner.

The possibility of providing alternative implementations for the operators of a semantic algebra was mentioned by Watt during his experimentation with ML as a semantic metalanguage [Wat84]. However, our MESS system is the first implementation generator which *enforces* the separation of the microsemantics from the semantics.

Very recently, Nielson and Nielson have described an approach to semantics directed compiler generation using the two level metalanguage TML which enforces the distinction between compile time and runtime domains [NiN86]. The composition of the two portions of a TML specification corresponds directly to the composition of macrosemantic and microsemantic definitions.

References

[Bir82] Bird, P. An implementation of a code generator specification language for table driven code generators. 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 a compiler generator based on denotational semantics. 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.

[ClF82] Clinger, W., Friedman, D. P., And Wand, M. A scheme for a higher-level semantic algebra. US-French Seminar on the Application of Algebra to Language Definition and Compilation, Fontainebleau, France, June 1982.

[Gan80] Ganapathi, M. Retargetable code generation and optimization using attribute grammars. Ph. D. Thesis, Tech. Rep. 406, Computer Science Department, University of Wisconsin-Madison, 1980.

[GlG78] Glanville, R. S., and Graham, S. A new method for compiler code generation. Conf. Rec. 5th Ann. ACM Symp. Principles of Programming Languages, Tucson, AZ, Jan. 1978.

[Gor79] Gordon, M. J. C. *The denotational description of programming languages: An introduction.* Springer-Verlag, New York, 1979.

[JoC82] Jones, N. D., and Christiansen, H. Control flow treatment in a simple semantics-directed compiler generator. Formal Description of Programming Concepts II, IFIP IC-2 Working Conference, North Holland, Amsterdam, 1982.

[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.

[LeP86a] Lee, P., and Pleban, U. F. The automatic generation of realistic compilers from high-level semantic descriptions: A progress report. Technical Report CRL-TR-13-86, The University of Michigan Computing Research Laboratory, June 1986.

[LeP86b] Lee, P., and Pleban, U. F. On the use of LISP in implementing denotational semantics. Proc. 1986 ACM Conf. LISP and Functional Programming, 233-248.

[Log86] *Logitech Modula-2 Compiler Manual (version 2.0).* Logitech, 1986.

[Man85] *Aztec C Development Package Manual (version 3.20e).* Manx, 1985.

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

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

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

[Mos82] Mosses, P. Abstract semantic algebras! In: D. Bjørner (Ed.), Formal description of programming concepts II. North Holland, Amsterdam, 1982, 63-88.

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

[NiN86] Nielson, H. R., and Nielson, F. Semantics directed compiling for functional languages. Proc. 1986 ACM Conf. LISP and Functional Programming, 249-257.

[Pau81] Paulson, L. A compiler generator for semantic grammars. Ph. D. Dissertation, Stanford University, December 1981.

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

[Ple84a] Pleban, U. Formal semantics and compiler generation. In: Morgenbrod, H., and Sammer, W. (Eds.) Programmierumgebungen und Compiler. Teubner-Verlag, Stuttgart, 1984, 145-161.

[Ple84b] Pleban, U. Compiler prototyping using formal semantics. Proc. SIGPLAN '84 Symp. Compiler Construction, SIGPLAN Notices 19, 6 (June 1984), 94-105.

[PlL86] Pleban, U. F., and Lee, P. High-level semantics: An integrated approach to programming language semantics and the specification of implementations (Summary). Submitted to the Third Workshop on the Mathematical Foundations of Programming Semantics. New Orleans, April 8-10, 1987.

[Roy86] Royer, V. Transformations in denotational semantics in semantics directed compiler generation. Proc. SIGPLAN '86 Symp. Compiler Construction, SIGPLAN Notices 21, 6 (June 1986), 68-73.

[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.

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

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

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

[Wan82] Wand, M. Deriving target code as a representation of continuation semantics. ACM TOPLAS 4, 3 (July 1982), 496-517.

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

[Wat82] Watt, D. A. Modular language definitions. Rep. CSC/82/R3, Computing Science Department, University of Glasgow, Oct. 1982.

[Wat84] Watt, D. A. Executable semantic descriptions. Rep. CSC/84/R2, Computing Science Department, University of Glasgow, Oct. 1984; also Software - Practice and Experience 16, 1 (Jan. 1986), 13-43.

BLESS
this
MESS