An Automatically Generated and Provably Correct Compiler for a Subset of Ada

Jens Palsberg
palsberg@daimi.aau.dk

Computer Science Department, Aarhus University
Ny Munkegade, DK-8000 Aarhus C, Denmark

January 1992

Abstract

We describe the automatic generation of a provably correct compiler for a non-trivial subset of Ada. The compiler is generated from an action semantic description; it emits absolute code for an abstract RISC machine language that currently is assembled into code for the SPARC and the HP Precision Architecture. The generated code is an order of magnitude better than what is produced by compilers generated by the classical systems of Mosses, Paulson, and Wand. The use of action semantics makes the processable language specification easy to read and pleasant to work with.

A reformatted version of this report (with only an excerpt of appendix B) is to be presented at *ICCL'92, Fourth IEEE International Conference on Computer Languages*, to be held 20–23 April, 1992, in San Francisco, California. Citations should refer to the Proceedings.

1 Introduction

The purpose of a language designer's workbench, envisioned by Pleban, is to drastically improve the language design process. The major components in such a workbench are:

- A *specification language* whose specifications are easily maintainable, and accessible without knowledge of the underlying theory; and
- A *compiler generator* that generates realistic compilers from such specifications.

With such a workbench, the language designer can:

- Document design decisions;
- Experiment with the new language after a change has been made; and
- Ship a compiler to programmers immediately after the design is finished.

This paper introduces another aspect to the notion of a language designer's workbench: *provable corretness*. Proving software correct is difficult in general, but if we can prove that compilers are correct, then an important class of errors is eliminated. We suggest that the compiler generator should produce compilers that are both realistic and provably correct.

We have taken a major step in this direction. We have designed, implemented, and proved the correctness of a compiler generator, called Cantor, that accepts action semantic descriptions of programming languages. The considered subset of action notation, see appendix A, is powerful enough to allow the specification of a non-trivial subset of Ada [5], called Mini-Ada, see appendix B. The generated compilers emit absolute code for an abstract RISC [40] machine language, which easily can be compiled into code for existing RISC processors. Currently, implementations exist for the SPARC [14] and the HP Precision Architecture [28].

The development of Cantor was guided by the following principles:

- Correctness is more important than efficiency; and

- Specification and proof must be completed before implementation begins.

As a result, on the positive side, the Cantor implementation was quickly produced, and only a handful of minor errors (that had been overlooked in the proof!) had to be corrected before the system worked. On the negative side, the generated compilers emit code that run at least two orders of magnitude slower than corresponding target programs produced by handwritten compilers. This is somewhat far from the goal of generating realistic compilers, but is still an improvement compared to the classical systems of Mosses, Paulson, and Wand where a slow-down of three orders of magnitude has been reported [11].

Action semantics was designed to allow accessible and maintainable descriptions of realistic programming languages. Our experiments with Cantor confirm that action semantic descriptions are easy to work with in practice. Future work on Cantor will attempt to improve speed without sacrificing provable correctness.

In the following section we examine the major previous approaches to compiler generation. In section 3 we outline the structure of the Cantor system, and we take a closer look at the generated Mini-Ada compiler. Finally, in section 4 we compare the performance of the generated Mini-Ada compiler with the standard C compilers on the SPARC and the HP Precision Architecture.

This paper summarizes the author's forthcoming PhD thesis [29], except the correctness proof. For an overview of our approach to correctness, see [30].

2 Previous Work

We will examine each of the previous approaches to compiler generation by focusing on:

- The *accessibility* and *maintainability* of the involved specifications;
- The *quality* of the generated compilers; and

- Whether *correctness* has been proved.

These criteria decide whether a system could be useful in a langauge designer's workbench.

Common to all of the approaches are that they choose a specific target language [32]. Ideally, the task is then to write and prove the correctness of a compiler for the involved specification language. Such a compiler can then be composed with a language definition to yield a correct compiler for the language, see figure 1. This approach is usually called *semantics-directed* compiler generation.

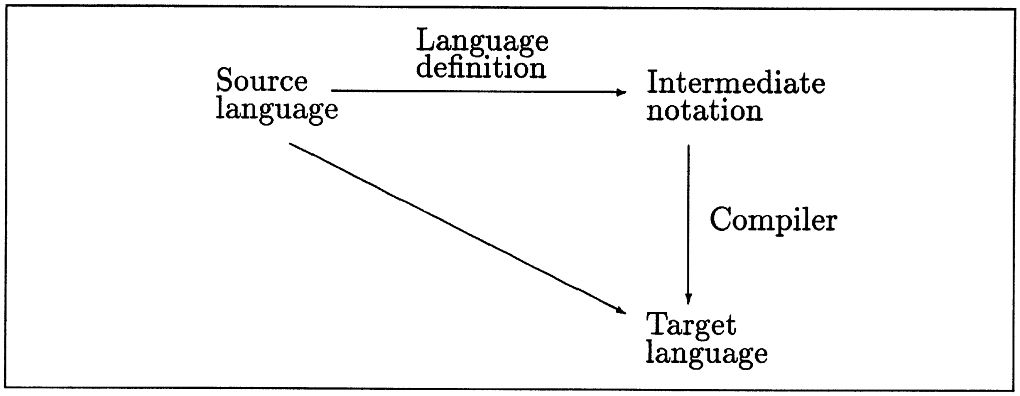

Figure 1: Semantics-directed compiler generation.

The traditional approach to compiler generation is based on *denotational semantics* [37]. Examples of existing compiler generators based on this idea include Mosses' Semantics Implementation System (SIS) [17], Paulson's Semantics Processor (PSP) [31, 32], and Wand's Semantic Prototyping System (SPS) [44]. Denotational semantics has achieved much popularity as a vehicle for theoretical studies, but it is also recognized to be neither flexible nor readable, see for example the discussions by Mosses [19], and Pleban and Lee [34]. The target programs produced by the classical systems have been reported to run at least three orders of magnitude slower than corresponding target programs produced by handwritten compilers [11]. None of these systems have been proved correct. In particular, even though SIS is based on a direct implementation of beta-reduction, then the implementation of that has not been proved correct. We conclude that the classics systems fail on all three points to be useful in a language designer's workbench.

A number of compiler generators have been built that produce compilers of a quality that compare well with commercially available compilers. Major examples are the CAT system of Schmidt and Völler [38, 39], the compiler generator of Kelsey and Hudak [10], and the Mess system of Pleban and Lee [33, 12, 35, 11]. These approaches are based on rather ad hoc specification languages, and, like the classical systems, they lack correctness proofs.

The CAT system is aimed at generating compilers for Pascal, C, Basic, Fortran, and Cobol. The specification language, called CAT, is a simplification of the union of all their syntactic constructs. This makes CAT itself into a high-level language which has its applicability as specification language limited to only little more than the five languages under consideration.

The compiler generator of Kelsey and Hudak has been used to generate compilers for Pascal, Basic, and Scheme. The specification language is a call-by-value lambda calculus with data and procedure constants and an implicit store. This makes the approach less general than the classical ones, in that it is biased towards a specific style of architecture.

Designer of the system	Specification language	Quality of generated compilers	Correctness Proof
Mosses	Denotational Semantics	Poor	No
Paulson	Denotational Semantics	Poor	No
Wand	Denotational Semantics	Poor	No
Schmidt and Völler	Amalgamation of five languages	Good	No
Kelsey and Hudak	Lambda notation with implicit store, etc.	Good	No
Pleban and Lee	High-level semantics	Good	No
Gomard and Jones	Denotational Semantics	Poor	Yes

Figure 2: Existing Compiler Generators.

The Mess system was created as a reaction to the lack of separation between conceptual analysis and model details that is found in the classical compiler generators. Instead of denotational semantics, the approach h to defining languages is *high-level* semantics. High-level semantics is compositional, but it does not have a standardized core notation, as does denotational semantics; it is rather a particular style of specification that is advocated.

This style involves a notion of *actions*, akin to and inspired by the actions found in precursors of action semantics. A high-level semantic definition involve essentially only compile-time objects; the run-time objects are then used in the definition of the notation for actions. This separation is the key to the success of the Mess system. It has been used to generate a compiler for a non-trivial imperative language.

The three realistic compiler generators trade generality for speed. It it not at all clear how to prove them correct, however, and in the case of the Mess system, such a proof must be given afresh for each new language because new actions often have to be introduced and defined. Work on compiler correctness does not seem to be of much help because it usually focuses on denotational semantics [13, 15, 41, 36, 27], algebraic variations hereof [3, 16, 42, 2, 18], structural operational semantics [6], or natural semantics [4].

We are aware of only one compiler generator that has been proved correct: the one obtained by self-application of the partial evaluator mix, see the paper by Gomard and Jones [7]. Unfortunately, the generated compilers emit code for the lambda calculus, thus leaving considerable compilation to be done. It remains to be seen if this approach will lead to the generation of compilers for conventional machine architectures.

A summary of the examination is given in figure 2. We will now consider the Cantor system which trades speed for correctness, but still produces better code than the classical systems of Mosses, Paulson, and Wand.

3 The Cantor System

Our compiler generator accepts action semantic descriptions. Action semantics is a framework for formal semantics of programming languages, developed by Mosses [19, 20, 21, 24, 25] and Watt [26, 45]. It is intended to allow useful semantic descriptions of realistic programing languages, and it is *compositional*, like denotational semantics. It differs from denotational semantics, however, in using semantic entities called *actions*, rather than higher-order functions.

We have designed a subset of action notation which is amenable to compilation and which we have given a natural semantics, by a systematic transformation of its structural operational semantics [25]. The syntax of this subset is given in appendix A together with a brief overview of some the principles behind action semantics. Appendix B presents a complete description of a subset of Ada, called Mini-Ada, featuring static typing, constants, variables, one-dimensional array-types, functions and procedures with in and in out (reference) parameters, various control structures, and the usual expressions. Note that the select construct in Mini-Ada can be used as a "case"-statement, and that also the input-output statements (read and write) are non-standard Ada. The Mini-Ada specification is a subset of one given by Mosses in his book [25]. (Readers who are unfamiliar with action semantics are not expected to understand the details in appendix B, despite the suggestiveness of the symbols used. See [25] for a full presentation of action semantics.)

In the following, we first give an overview of the structure of Cantor and the generated Mini-Ada compiler. We then discuss the machine language used, and finally we take a closer look at how to compile actions in a provably correct fashion.

3.1 Overview

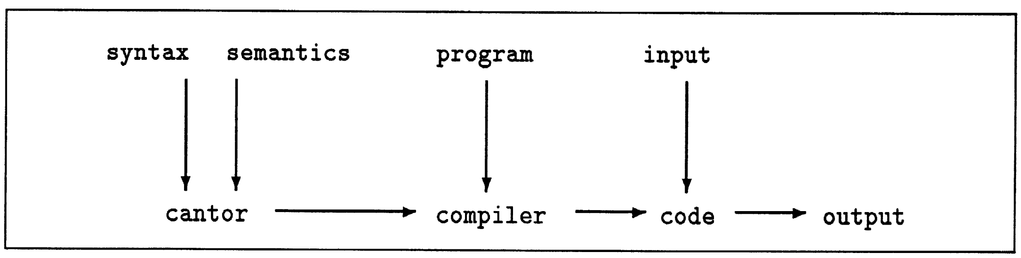

Figure 3: The Cantor system.

The Cantor system has the structure shown in figure 3. In practice, a session with Cantor looks as follows on the screen:

```
cantor syntax semantics compiler
compiler program code
code input output
```

The compiler generator `cantor` is written in Perl [43], and the generated compilers are written in Scheme [1]. Examples of a syntax and a semantics are given in appendix B; it is the LaTeX source of the appendix that is processed by `cantor`. The generated compiler contains a syntax checker, a program-to-action transformer, the action compiler described above, and finally a Pseudo SPARC assembler that currently can emit code for the SPARC and the HP Precision Architecture. The input file is a sequence of integers, as is the output file.

3.2 An Abstract RISC Machine Language

The machine language is patterned after the SPARC architecture; it is called Pseudo SPARC. It contains 14 instructions that operate on a model of the SPARC machine state, including status-bits, register-windows, main memory, etc. The only data manipulated are integers, thus making the language more realistic than those considered in most previous compiler proofs. It contains two idealizations, however, as follows:

- **Unbounded word and memory size:** The data values are *unbounded* integers and this requires unbounded word size. We also assume that the program and memory sizes, the number of of registers in a register window, and the number of register windows me unbounded.
- **Read-only code:** The program is placed separately, not in 'memory'. This implies that code will not be overwritten, and that data will not be "executed".

Furthermore, we do not model delay slots. These idealizations simplify the correctness proof considerably, but they may be removed in future work, using the technique of Joyce [9, 8].

Figure 4 shows the 14 Pseudo SPARC instructions and how they (approximately) can be expanded to real SPARC instructions. Pseudo SPARC instructions can also be expanded to instructions for the HP Precision Architecture, though with a little more difficulty.

3.3 Compiling Action Notation

The compiler from action notation to Pseudo SPARC machine code proceeds in two passes:

1. Type analysis and calculation of code size; and
2. Code generation.

For each pass there is a function defined for every syntactic category. Those defined for 'Act' have the following signatures:

a-count _ _ _ :: Act, data-type, symbol-table →
    (natural, truth-value, data-type,
      truth-value, data-type, block) .

perform _ _ _ _ _ _ _ _ _ _ _ _ ::
    Act, data-types, general-register, frozen,
    symbol-table, cleanup, cleanup, cleanup,
    linenumber, linenumber-complete,
    linenumber-escape, linenumber-fail →
    (program, general-register, general-register) .

Since action notation contains unusual constructs, the definition of the type analysis and code generation employ unusual techniques, though not very difficult. For example, the definition of 'perform' requires as argument both the desired start-address ('linenumber') of the code to be generated, but also addresses of where to jump to, should the performance complete ('linenumber-complete'), escape ('linenumber-escape'), or fail ('linenumber-fail'). These addresses are calculated using 'a-count' which, in addition to type analysis, calculates the size of the code to be generated.

As an example of how the compiler works, see the following excerpt from the compiling specification.

(1) d-count D h d = (n:natural, truth-value-type)
⇒ a-count ⟦ "check" D:Dependent ⟧ h d = ac-state
    sum(n, 2, e-size, 12) true () false () empty-list .

Pseudo SPARC	Real SPARC
skip	`sub %g0, %g0, %g0`
jump Z	`jmpl` Z`, Xg0`
branchequal Z	`be` Z
branchlessthan Z	`bneg` Z
call	`jmpl global, %r8`
return	`jmpl %r8+8, %g0`
store $R1$ in $R2$ Z P	`st` $R1$`,` $R2+Z+P$
load $R1$ Z P into $R2$	`ld` $R1+Z+P$`,` $R2$
storeregisters	save
load registers	restore
move RI to R	`or %g0,` RI`,` R
move sum R RI to R'	add R, RI, R'
move difference R RI to R'	sub R, RI, R'
compare R with RI	subcc R, RI, %g0

Figure 4: The Pseudo SPARC machine language.

(1) d-count D h d $=$ (n:natural, truth-value-type) ;
(2) $l' =$ sum(l, n) ;
(3) $l'' =$ sum(l', 2, e-size) ;
(4) evaluate D h a f d l sum(l'',6) $=$
(p:program, r:general-register)
$\Rightarrow$ perform $[\![$ "check" D "Dependent" $]\!]$ h a f d
u_n u_e u_f l l_n l_e l_f $=$ a-state overlay(
p,
map of sum(l',0) to (compare r with 0),
map of sum(l',1) to (branchequal sum(l'',6)),
empty-list-code r sum(l',2),
putcommit l'' 0,
finalize sum(l'',3) u_n 0 l_n,
putcommit sum(l'',6) 0,
finalize sum(l'',9) u_f 2 l_f)

r a .

The first definition calculates the size of the code generated by the secand definition. It also does the type-checking. The meaning of the action 'check D' is to check whether D evaluates to true or false, and it should then "complete" or "fail", accordingly. The generated code first computes the result of D, and then it does a branchequal, as expected. (We represent true as 1 and false as 0.) This is not all, however. Because of the generality of action notation a lot of additional code is also generated. We will not explain the details, as it requires an intimate knowledge of the semantics of action notation, but simply note that a commonly found action such as 'check (it is true)' yields 37 lines of code. It should be noted, though, that it is this clear structure of the code that made the correctness proof manageable.

Our approach to correctness can be summarized as follows:

1. Give a natural semantics to both action notation and the abstract RISC machine language.
2. Make the compiling of action notation simple; and
3. Use a variation of Despeyroux's proof technique [4].

All specifications are given using unified algebras, an algebraic specification framework developed by Mosses [23, 21, 22]. This includes the semantics of action notation (13 pages), the semantics of the machine language (6 pages), the compiler (36 pages), and various auxiliary notation (14 pages). The correctness statement, including various lemmas but without proofs, takes 28 pages. Putting further sophistication into the compiler will add significantly to these page counts. We feel that the size alone of the specifications calls for automatic proof checking. Recent attempts to automatically check a compiler correctness proof are reported by Young [46] and Joyce [9, 8]. For now, however, we leave the automatic checking of the Cantor correctness proof to future work and turn to a performance evaluation.

4 Performance Evaluation

The Mini-Ada action semantics in appendix B has been the primary benchmark in our experiments with the Cantor system.

- Generating the Mini-Ada compiler takes 9 seconds.

We have used this compiler to translate a number of benchmark programs, described in figure 5. The sieve, euclid, and fib programs contain a main loop that allows iterating the computation. This will be practical when we later compare the object code emitted by the Mini-Ada compiler with that emitted by handwritten compilers.

bubble: Bubblesorts a number of integers (50 lines).

sieve: Performs the sieve of Erathosthenes prime number generator (30 lines).

euclid: Computes the greatest common divisor of two numbers using Euclid's algorithm (20 lines).

fib: Computes the 56'th Fibonacci number (30 lines).

Figure 5: The Mini-Ada benchmark programs.

The number of Pseudo SPARC instructions emitted for each benchmark program is given in figure 6. When the Pseudo SPARC code is compiled to code for the SPARC, then the size is approximately doubled. A slightly worse blow-up is obtained when compiling to the HP Precision Architecture.

No. of Pseudo SPARC instructions generated:

bubble:	16697	euclid :	7386
sieve :	12096	fib :	9095

Figure 6: Object code size.

Unfortunately, we have no access to an Ada compiler that generates code for either of the two architectures that we consider. Instead, we have made comparison with the standard C compiler for those architectures. It is perhaps unfair to compare Ada and C, but we still believe that using the C compiler gives a good indication of the capabilities of Cantor. We expect that the C compilers generates better code than potential Ada compilers. Hence, when we compute the slow-down compared to C, we will take it as an upper bound of the slow-down compared to Ada. We of course had to rewrite the Mini-Ada programs slightly to get them accepted by the C compilers. Since the constructs in C are less general than those in Ada, we expect a significantly better performance of the C-generated code, than what could be expected from Ada-generated code.

	C	C^{opt}	Mini-Ada
bubble	1.0	2.2	542
sieve	1.2	2.1	377
euclid	1.1	1.6	136
fib	1.1	1.7	210

Figure 7: Compile times.

Figure 7 shows the compile time in seconds when using the C compiler, the C compiler with maximal optimization switched on, and the Cantor-generated Mini-Ada compiler. The timings in this figure were recorded on the SPARC, as the compilers run almost equally fast on the HP. The timings indicate that the Cantor system is rather tedious to work with in practice. We plan to rewrite the action compiler in C instead of Scheme, to get acceptable compile times.

	C	C^{opt}	Mini-Ada	Slow-down
bubble	4.4 (1000 numbers)	2.1 (1000 numbers)	0.9 (37 numbers)	149
sieve	1.3 (400 itera.)	0.4 (400 itera.)	1.2 (1 itera.)	369
euclid	5.4 (30000 itera.)	0.9 (30000 itera.)	0.8 (30 itera.)	148
fib	1.2 (10000 itera.)	0.2 (10000 itera.)	0.8 (36 itera.)	185

Figure 8: Object code execution time on the SPARC.

Figures 8 and 9 show the object code execution time in seconds for the benchmark programs. They also show the estimated slow-down when using the Mini-Ada compiler, compared to the C compiler *without* optimization.

	C	C^{opt}	Mini-Ada	Slow-down
bubble	7.2 (1000 numbers)	4.7 (1000 numbers)	4.3 (37 numbers)	436
sieve	1.2 (400 itera.)	0.4 (400 itera.)	4.5 (1 itera.)	1500
euclid	4.5 (30000 itera.)	4.4 (30000 itera.)	2.7 (30 itera.)	600
fib	1.1 (10000 itera.)	0.5 (10000 itera.)	3.9 (36 itera.)	985

Figure 9: Object code execution time on the HP Precision Architecture.

The slow-down factors were computed by simple extrapolation. The figures indicate, unsurprisingly, that the Mini-Ada-generated code runs faster on the SPARC than on the HP. This is because the Pseudo SPARC machine language was designed to match the SPARC instructions, not the HP instructions. Thus, more code is generated for each Pseudo SPARC instruction when compiling to the HP.

The performance of the object code is most fairly compared on the SPARC. Taking the differences of C and Ada into account, we conclude that the object code run at least two orders of magnitude slower than corresponding code produced by handwritten Ada compilers.

5 Conclusion

We have taken a step towards the construction of a provably correct implementation of a practically useful language designer's workbench. We have illustrated our approach on a non-trivial subset of Ada, hoping to demonstrate that such a workbench could have been a helpful tool during the design of Ada.

While being provably correct, our compiler generator still generates significantly better code than the classical systems of Mosses, Paulson, and Wand. Future work may take four directions:

- **Better object code:** We will build in more compile time analysis, to improve the code generator.

- **Completely realistic target language:** We will define and use a target language without the idealizations discussed in this paper.

- **Faster compiler:** We will rewrite the action compiler in C instead of Scheme, to get acceptable compile times.

- **Automatic proof check:** We will exploit recent advances in automatic proof checking to obtain a very trustworthy system.

We believe that a provably correct and practically useful language designer's workbench is a realistic possibility.

Acknowledgement. This work has been supported in part by the Danish Research Council under the DART Project (5.21.08.03). The author thanks Peter Mosses and Michael Schwartzbach for helpful comments on a draft of the paper. The author also thanks Peter Ørbæk for implementing the Cantor system.

A Action Notation

needs: **Data Notation/Numbers/Naturals .**
introduces: token .
grammar:

Act	=	"complete" | "escape" | "fail" |
		"commit" | "diverge" | "regive" |
		⟦ "give" Dependent ⟧ | ⟦ "check" Dependent ⟧ |
		⟦ "bind" token "to" Dependent ⟧ |
		⟦ "store" Dependent "in" Dependent ⟧ |
		⟦ "allocate" ("truth-value" | "integer") "cell" ⟧ |
		⟦ "batch-send" Dependent ⟧ | ⟦ "batch-receive" "an" "integer" ⟧ |
		⟦ "enact" "application" Dependent "to" Tuple ⟧ |
		⟦ "indivisibly" Act ⟧ | ⟦ "unfolding" Unf ⟧ | ⟦ Act Infix Act ⟧ |
		⟦ ⟦ "furthermore" Act ⟧ ("hence" | "thence") Act ⟧ .
Unf	=	⟦ Act Infix Unf ⟧ | ⟦ Unf "or" Act ⟧ | "unfold" .
Tuple	=	"()" | Dependent | ⟦ Tuple "," Tuple ⟧ | "them" .
Dependent	=	"true" | "false" | natural |
		⟦ "empty-list" "&" "[" Type "]" "list" ⟧ |
		⟦ "closure" "abstraction" "of" Act "&"
		"[" "perhaps" "using" Data "]" "act" ⟧ |
		⟦ Unary Dependent ⟧ | ⟦ Binary "(" Dependent "," Dependent ")" ⟧ |
		⟦ Dependent ("is" | ⟦ "is" "less" "than" ⟧) Dependent ⟧ |
		⟦ "component#" Dependent "items" Dependent ⟧ |
		"it" | ⟦ "the" "given" Datum "#" natural ⟧ |
		⟦ "the" Datum "bound" "to" token ⟧ |
		⟦ "the" Datum "stored" "in" Dependent ⟧ |
		⟦ "(" Dependent ")" ⟧
Infix	=	⟦ "and" "then" ⟧ | "then" | "before" | "trap" | "or" .
Unary	=	"not" | "negation" | ⟦ "list" "of" ⟧ | "head" | "tail" .
Binary	=	"both" | "either" | "sum" | "difference" | "concatenation" .
Datum	=	"datum" | "cell" | "abstraction " | "list" |
		⟦ Datum "|" Datum ⟧ | Type .
Data	=	"()" | Type | ⟦ Data "," Data ⟧ .
Type	=	"truth-value" | "integer" |
		⟦ "truth+alue" "cell" ⟧ | ⟦ "integer" "cell" ⟧ |
		⟦ "[" Type "]" "list" ⟧ .

A.1 Action Principles

Action notation is designed to allow comprehensible and accessible descriptions of programming languages. Action semantic descriptions scale up smoothly from small example languages to realistic languages, and they can make widespread reuse of action semantic descriptions of related languages.

Actions reflect the gradual, stepwise nature of computation. A performance of an action, which may be part of an enclosing action, either

- *completes*, corresponding to normal termination (the performance of the enclosing action proceeds normally); or
- *escapes*, corresponding to exceptional termination (the enclosing action is skipped until the escape is trapped); or
- *fails*, corresponding to abandoning the performance of an action (the enclosing action performs an alternative action, if there is one, otherwise it fails too); or
- *diverges*, corresponding to nontermination (the enclosing action also diverges).

The information processed by action performance may be classified according to how far it tends to be propagated, as follows:

- *transient*: tuples of data, corresponding to intermediate results;
- *scoped*: bindings of tokens to data, corresponding to symbol tables;
- *stable*: data stored in cells, corresponding to the values assigned to variables;
- *permanent*: data communicated between distributed actions.

Transient information is made available to an action for immediate use. Scoped information, in contrast, may generally be referred to throughout an entire action, although it may also be hidden temporarily. Stable information can be changed, but not hidden, in the action, and it persists until explicitly destroyed. Permanent information cannot even be changed, merely augmented.

When an action is performed, transient information is given only on completion or escape, and scoped information is produced only on completion. In contrast, changes to stable information and extensions to permanent information are made *during* action performance, and are unaffected by subsequent divergence or failure.

Our subset of action notation omits all notation for communication. Instead, the ad hoc constructs 'batch-send' and 'batch-receive' allow a primitive form of communication with batch-files, as in standard Pascal.

The information processed by actions consist of items of *data*, organized in structures that give access to the individual items. Data can include various familiar mathematical entities, such as truth-values, integers, and lists. Actions themselves are not data, but they can be incorporated in so-called abstractions, which are data, and subsequently 'enacted' back into actions.

Dependent data are entities that can be *evaluated* to yield data during action performance. The data yielded may depend on the current information, i.e., the given transients, the received bindings, and the current state of the storage and batch-files. Evaluation cannot affect the current information. Data is a special case of dependent data, and it always yields itself when evaluated.

B Mini-Ada Action Semantics

B.1 Abstract Syntax

grammar:

Program = ⟦ Declarations Identifier ⟧ .

Declarations = ⟦ Declarations Declarations ⟧ |
⟦ Identifier “:” “constant” “:=” Expression “;” ⟧ |
⟦ Identifier “:” Nominator “;” ⟧ |
⟦ Identifier “:” Nominator “:=” Expression “;” ⟧ |
⟦ “type” Identifier “is” “array”
“(” “0” “..” Expression “)” “of” Primitive “;” ⟧ |
⟦ “function” Identifier “return” “integer” “is” Block “;” ⟧ |
⟦ “function” Identifier “(” Formals-In “)”
“return” “integer” “is” Block “;” ⟧ |
⟦ “procedure” Identifier “is” Block “;” ⟧ |
⟦ “procedure” Identifier “(” Formals “)” “is” Block “;” ⟧ .

Formals = ⟦ Formal “;” Formals ⟧ | Formal .

Formal = ⟦ Identifier “:” “in” “out” “integer” ⟧ .

Formals-In = ⟦ Formal-In “;” Formals-In ⟧ | Formal-In .

Formal-In = ⟦ Identifier “:” “integer” ⟧ .

Nominator = Primitive | Identifier .

Primitive = “boolean” | “integer” .

Statements = ⟦ Statements Statements ⟧ |
⟦ “null” “;” ⟧ |
⟦ Name “:=” Expression “;” ⟧ |
⟦ “if” Expression “then” Statements “end” “if” “;” ⟧ |
⟦ “if” Expression “then” Statements
“else” Statements “end” “if” “;” ⟧ |
⟦ “select” Alternatives “end” “select” “;” ⟧ |
⟦ “select” Alternatives “else” Statements “end” “select” “;” ⟧ |
⟦ “loop” Statements “end” “loop” “;” ⟧ |
⟦ “while” Expression “loop” Statements “end” “loop” “;” ⟧ |
⟦ “exit” “;” ⟧ |
⟦ “begin” Statements “end” “;” ⟧ |
⟦ “declare” Declarations “begin” Statements “end” “;” ⟧ |
⟦ Identifier “;” ⟧ |

⟦ Identifier “(” Names “)” “;” ⟧ |
⟦ “return” “;” ⟧ |
⟦ “return” Expression “;” ⟧ |
⟦ “write” Expression “;” ⟧ |
⟦ “read” Name “;” ⟧ .
Block = ⟦ “begin” Statements “end” ⟧ |
⟦ Declarations “begin” Statements “end” ⟧ .
Alternatives = Statements |
⟦ “when” Expression “=>” Statements ⟧ |
⟦ Alternatives “or” Alternatives ⟧ .
Names = Name | ⟦ Names “;” Names ⟧ .
Name = Identifier | ⟦ Identifier “(” Expressions ”)” ⟧ .
Expressions = Expression | ⟦ Expressions “;” Expressions ⟧ .
Expression = “true” | “false” | Integer | Name |
⟦ “(” Expression “)” ⟧ |
⟦ “not” Expression ⟧ |
⟦ Expression Binary-Operator Expression ⟧ |
⟦ Expression Control-Operator Expression ⟧ .
Binary-Operator = “+” | “−” | “=” | “/ =” | “<” | “<=” |
“>” | “>=”| “and” ” | “or” ” | “xor” .
Control-Operator = ⟦ “and” “then” ⟧ | ⟦ “or” “else” ⟧ .
Integer = natural | ⟦ “−” natural ⟧ .
Identifier = token .

B.2 Semantic Entities

B.2.1 Items

introduces: item , parameter-less-procedure , parameterized-procedure ,
parameter-less-function , parameterized-function ,
non-abstraction , escape-reason , exit , function-return , procedure-return ,
there-is-given-a n-exit , there-is-given-a-return ,
there-is-given-a-procedure-return , err .

item = truth-value | integer .
parameter-less-procedure = abstraction .
parameterized-procedure = abstraction .
parameter-less-function = abstraction .

parameterized-function = abstraction .
non-abstraction = item | cell | list .
escape-reason = [integer] list .
exit = list of 0 .
function-return = [integer] list .
procedure-return = list of 2 .
there-is-given-an-exit = (component# 1 items it) is 0 .
there-is-given-a-return =
either((component# 1 items it) is 1 , (component# 1 items it) is 2) .
there-is-given-a-procedure-return = (component# 1 items it) is 2 .
err = commit and then fail .

B.2.2 Closures

introduces: function-return-of _ , returned-value-of _ ,
parameter-less-closure _ ,
parameterized-function-closure _ , parameterized-procedure-closure _ .

- function-return-of _ :: integer → [integer] list .
- returned-value-of _ :: [integer] list → integer .
- parameter-less-closure _ :: act → dependent datum .
- parameterized-function-closure _ :: act → dependent datum .
- parameterized-procedure-closure _ :: act → dependent datum .

function-return-of i:integer = concatenation(list of 1, list of i) .
returned-value-of l:[integer] list = component# 2 items l .
parameter-less-closure A:act = closure abstraction of A & [perhaps using ()] act .
parameterized-function-closure A:act =
closure abstraction of A & [perhaps using [integer] list] act .
parameterized-procedure-closure A:act =
closure abstraction of A & [perhaps using [integer cell] list] act .

B.3 Semantic Functions

introduces: run _ , elaborate _ , actualize-formals _ , actualize-formal _ ,
actualize-formals-in _ , actualize-formal-in _ ,
allocate-for _ , allocate-for-primitive _ ,
execute _ , execute-block _ , exhaust _ ,

multi-investigate _ , investigate _ ,
multi-evaluate _ , evaluate _ ,
the-binary-operation-result-of _ ,
the-control-operation-completion-of _ ,
integer-value _ , id _ .

B.3.1 Program

• run _ :: Program → act .
run ⟦D:Declarations I:dentifier ⟧ =

| furthermore elaborate D
hence
| enact application (the parameter-less-procedure bound to id I) to () .

B.3.2 Declarations

• elaborate _ :: Declarations → act .
elaborate ⟦D_1:Declarations D_2:Declarations ⟧ = elaborate D_1 before elaborate D_2 .
elaborate ⟦I:Identifier ":" "constant" ":=" E:Expression ";" ⟧ =

evaluate E then bind id I to it .

elaborate ⟦I:Identifier ":" N:Nominator ";" ⟧ =

allocate-for N then bind id I to it .

elaborate ⟦I:Identifier ":" N:Nominator ":=" E:Expression ";" ⟧ =

| allocate-for N and then evaluate E
then
| store the given item #2 in the given cell #1 and then
| bind id I to the given datum #1 .

elaborate ⟦ "type" I:Identifier "is" "array"
"(" "0" ".." E: Expression ")" "of" "boolean" ";" ⟧ =
bind id I to parameter-less-closure

give empty-list & [truth-value cell] list and then
evaluate E then give sum(it, 1)
then
unfolding
check the given integer #2 is 0 and then
give the given list #1
or
regive and then
allocate truth-value cell
then
give concatenation(list of the given truth-value cell #3, the given list #1)
and then
give difference(the given integer #2, 1)
then
unfold .

elaborate ⟦ "type" I:Identifier "is" "array"
"(" "0" ".. " E:Expression ")" "of" "integer" ";" ⟧ =
bind id I to parameter-less-closure
give empty-list & [integer cell] list and then
evaluate E then give sum(it, 1)
then
unfolding
check the given integer #2 is 0 and then
give the given list #1
or
regive and then
allocate integer cell
then
give concatenation(list of the given integer cell #3, the given list #1)
and then
give difference(the given integer #2, 1)
then
unfold .

elaborate ⟦ "function" I:Identifier "return" "integer" "is" B:Block ";" ⟧ =

bind id I to parameter-less-closure
| | execute-block B and then err
| trap give returned-value-of the given function-return #1 .

elaborate ⟦ "function" I:Identifier "(" F:Formals-In ")"
"return" "integer" "is" B:Block ";" ⟧ =

bind id I to parameterized-function-closure
| furthermore actualize-formals-in F thence
| | | execute-block B and then err
| | trap give returned-value-of the given function-return #1 .

elaborate ⟦ "procedure" I:Identifier "is" B:Block ";" ⟧ =

bind id I to parameter-less-closure
| execute-block B
| trap check there-is-given-a-procedure-return .

elaborate ⟦ "procedure" I:Identifier "(" F:Formals ")" "is" B:Block ";" ⟧ =

bind id I to parameterized-procedure-closure
| furthermore actualize-formals F thence
| | execute-block B
| | trap check there-is-given-a-procedure-return .

B.3.3 Formals

• actualize-formals _ :: Formals → act .

actualize-formals ⟦ F_1 :Formal ";" F_2 :Formals ⟧ =
| give head the given list #1 then actualize-formal F_1
before
| give tail the given list #1 then actualize=formals F_2 .

actualize-formals F:Formal =
give head the given list #1 then actualize-formal F .

B.3.4 Formal

• actualize-formals _ :: Formals → act .

actualize-formals ⟦ I :Identifier ":" "in" "out" "integer" ⟧ =
bind id I to the given integer cell #1 .

B.3.5 Formals-In

• actualize-formals-in _ :: Formals-In → act .
actualize-formals-in ⟦ F_1:Formal-In ";" F_2:Formals-In ⟧ =

| give head the given list #1 then actualize-formal F_1
before
| give tail the given list #1 then actualize-formals-in F_2 .

actualize-formals-in F:Formal-In =

give head the given list #1 then actualize-formal-in F .

B.3.6 Formal-In

• actualize-formal-in _ :: Formal-In → act .
actualize-formal-in ⟦ I :Identifier ":" "integer" ⟧ =

bind id I to the given integer cell #1 .

B.3.7 Nominator

• allocate-for _ :: Nominator → act .
allocate-for P:Primitive = allocate-for-primitive P .
allocate-for I:Identifier =

enact application (the abstraction bound to id I) to () .

B.3.8 Primitive

• allocate-for-primitive _ :: Primitive → act .
allocate-for-primitive "boolean" = allocate truth-value cell .
allocate-for-primitive "integer" = allocate integer cell .

B.3.9 Statements

• execute _ :: Statements → act .
execute ⟦S_1:Statements S_2:Statements ⟧ = execute S_1 and then execute S_2 .
execute ⟦ "null" ";" ⟧ = complete .

execute ⟦N:Name ":=" E:Expression ⟧ =

| investigate N and then evaluate E

then store the given item #2 in the given cell #1 .

execute

⟦ "if" E:Expression "then" S:Statements "end" "if" ";" ⟧ =

evaluate E then

| | check (it is true) and then execute S

| or

| | check (it is false) .

execute ⟦ "if" E:Expression "then" S_1:Statements "else" S_2:Statements "end" "if" ";" ⟧ =

evaluate E then

| | check (it is true) and then execute S_1

| or

| | check (it is false) and then execute S_2 .

execute ⟦ "select" A:Alternatives "end" "select" ";" ⟧ =

| exhaust A

| trap

| | enact application the given abstraction #1 to () .

execute ⟦ "select" A:Alternatives "else" S:Statements "end" "select" ";" ⟧ =

| exhaust A and then

| | | give parameter-less-closure execute S

| | then escape

trap

| enact application the given abstraction #1 to () .

execute ⟦ "loop" S:Statements "end" "loop" ";" ⟧ =

| unfolding

| | execute S and then unfold

trap

| check there-is-given-an-exit

or

| check there-is-given-a-return and then escape .

execute ⟦ “while” E:Expression “loop” S:Statements “end” “loop” “;” ⟧ =

| unfolding
| | evaluate E then
| | | | check (it is true) and then execute S and then unfold
| | | or check (it is false)
trap
| | check there-is-given-an-exit
| or
| | check thereais-given-a-return and then escape .

execute ⟦ “exit” “;” ⟧ = give exit then escape .
execute ⟦ “begin” S:Statements “end” “;” ⟧ = execute S .
execute ⟦ “declare” D:Declarations “begin” S:Statements “end” “;” ⟧ =

furthermore elaborate D hence
| execute S .

execute ⟦I:Identifier “;” ⟧ =

enact application the parameter-less-procedure bound to id I to () .

execute ⟦I:Identifier “(” N:Names “)” “;” ⟧ =

| give the parameterized-procedure bound to id I and then
| multi-investigate N
then
| enact application the given abstraction #1 to the given list #2 .

execute ⟦ “return” “;” ⟧ = give procedure-return then escape .
execute ⟦ “return” E:Expression “;” ⟧ =

evaluate E then
give function-return-of it then
escape .

execute ⟦ “write” E:Expression “;” ⟧ =

evaluate E then batch-send it .

execute ⟦ “read” N:Name “;” ⟧ =

| batch-receive an integer and then investigate N
then store the given integer #1 in the given integer cell #2 .

B.3.10 Block

• execute-block _ :: Block → act .
execute-block ⟦ "begin" S:Statements "end" ⟧ = execute S .
execute-block ⟦ D:Declarations "begin" S:Statements "end" ⟧ =
    furthermore elaborate D hence
    | execute S .

B.3.11 Alternatives

• exhaust _ :: Alternatives → act .
exhaust S:Statements =
    | give parameter-less-closure execute S
    then escape .
exhaust ⟦ "when" E:Expression "=>" S:Statements ⟧ =
    evaluate E then
    | | | check (it is true) then
    | | | give parameter-less-closure execute S
    | | then escape
    | or check (it is false) .
exhaust ⟦A_1:Alternatives "or" A_2:Alternatives ⟧ =
    exhaust A_1 and then exhaust A_2 .

B.3.12 Names

• multi-investigate _ :: Names → act .
multi-investigate N:Name =
    investigate N then give list of it .
multi-investigate ⟦N_1:Names ";" N_2:Names ⟧ =
    | multi-investigate N_1 and then multi-investigate N_2
    then give concatenation(the given list #1, the given list #2) .

B.3.13 Name

• investigate _ :: Name → act .
investigate I:Identifier =

give the datum bound to id I then
| give the given non-abstraction #1 or
| enact application the given parameter-less-function #1 to () .

investigate ⟦I:Identifier "(" E:Expressions ")" ⟧ =

| give the datum bound to id I and then
| multi-evaluate E
then
| | | give the given list #1 and then
| | | give head the given [integer] list #2
| | then
| | | give component# sum(the given integer #2, 1) items (the given list #1)
| or
| | enact application (the given parameterized-function #1) to (the given list #2) .

B.3.14 Expressions

• multi-evaluate _ :: Expressions → act .
multi-evaluate E:Expression = evaluate E then give list of it .
multi-evaluate ⟦E_1:Expressions ";" E_2:Expressions ⟧ =

| multi-evaluate E_1 and then multi-evaluate E_2
then give concatenation(the given list #1, the given list #2) .

B.3.15 Expression

• evaluate _ :: Expression → act .
evaluate "true" = give true .
evaluate "false" = give false .
evaluate i:Integer = give integer-value i .
evaluate N:Name =
investigate N then
| give the given item #1 or
| give the item stored in the given cell #1 .

evaluate $[\![$ “(” E:Expression “)” $]\!]$ = evaluate E .
evaluate $[\![$ “not” E:Expression $]\!]$ = evaluate E then give not it .
evaluate $[\![E_1$:Expression O:Binary-Operator E_2:Expression $]\!]$ =
    evaluate E_1 and then evaluate E_2
    then give the-binary-operation-result-of O .
evaluate $[\![E_1$:Expression O:ControLOperator E_2:Expression $]\!]$ =
    evaluate E_1 then
        check the-control-operation-completion-of O and then
        give the given truth-value #1
    or
        check not the-control-operation-completion-of O then
        evaluate E_2 .

B.3.16 Binary-Operator

• the-binary-operation-result-of _ :: Binary-Operator → dependent datum .
the-binary-operation-result-of “+” =
    sum(the given integer #1, the given integer #2) .
the-binary-operation-result-of “−” =
    difference(the given integer #1, the given integer #2) .
the-binary-operation-result-of “=” =
    (the given item #1) is (the given item #2) .
the-binary-operation-result-of “/ =” =
    not ((the given item #1) is (the given item #2)) .
the-binary-operation-result-of “<” =
    (the given integer #1) is less than (the given integer #2) .
the-binary-operation-result-of “<=” =
    not ((the given integer #2) is less than (the given integer #1)) .
the-binary-operation-result-of “>” =
    (the given integer #2) is less than (the given integer #1) .
the-binary-operation-result-of “>=” =
    not ((the given integer #1) is less than (the given integer #2)) .
the-binary-operation-result-of “and” =
    both(the given truth-value #1, the given truth-value #2) .
the-binary-operation-result-of “or” =
    either(the given truth-value #1, the given truth-value #2) .
the-binary-operation-result-of “xor” =
    not ((the given truth-value #1) is (the given truth-value #2)) .

B.3.17 Control-Operator

• the-control-operation-completion-of _ :: Control-Operator → dependent datum.

the-control-operation-completion-of ⟦ "and" "then" ⟧ =

(the given truth-value #1) is false .

the-control-operation-completion-of ⟦ "or" "else" ⟧ =

(the given truth-value #1) is true .

B.3.18 Integer

• integer-value _ :: Integer → integer .
integer-value n:natural = n .
integer-value ⟦ "−" n:natural ⟧ = negation n .

B.3.19 Identifier

• id _:: Identifier → token .
id k:token = k .

References

[1] Harald Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Rudolf Berghammer, Herbert Ehler, and Hans Zierer. Towards an algebraic specification of code generation. *Science of Computer Programming*, 11:45–63, 1988.

[3] Rod M. Burstall and Peter J. Landin. Programs and their proofs: an algebraic approach. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence, Vol. 4*, pages 17–43. Edinburgh University Press, 1969.

[4] Joëlle Despeyroux. Proof of translation in natural semantics. In *LICS'86, First Symposium on Logic in Computer Science*, June 1986.

[5] Jean D. Ichbiah et al. *Reference Manuul for the Ada Programming Language.* US DoD, July 1982.

[6] Anders Gammelgaard and Flemming Nielson. Verification of the level 0 compiling specification. Technical report, Department of Computer Science, Aarhus University, July 1990.

[7] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[8] Jeffrey J. Joyce. Totally verified systems: Linking verified software to verified hardware. In *Proc. Hardware Specification, Verification und Synthesis: Mathmatical Aspects*, July 1989.

[9] Jeffrey J. Joyce. A verified compiler for a verified microprocessor. Technical report, University of Cambridge, Computer Laboratory, England, March 1989.

[10] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Sixteenth Symposium on Principles of Programming Languages.* ACM Press, January 1989.

[11] Peter Lee. *Realistic Compiler Generation.* MIT Press, 1989.

[12] Peter Lee and Uwe F. Pleban. A realistic compiler generator based on high-level semantics. In *Fourteenth Symposium on Principles of Programming Languages*, pages 284–295. ACM Press, January 1987.

[13] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In *Proc. Symposium in Applied Mathematics of the American Mathmatical Society*, April 1966.

[14] Sun Microsystems. A RISC tutorial. Technical Report 800-1795-10, revision A, May 1988.

[15] Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, 1976.

[16] Francis Lockwood Morris. Advice on structuring compilers and proving them correct. In *Symposium on Principles of Programming Languages*, pages 144–152. ACM Press, October 1973.

[17] Peter D. Mosses. SIS—semantics implementation system. Technical Report Daimi MD–30, Computer Science Department, Aarhus University, 1979.

[18] Peter D. Mosses. A constructive approach to compiler correctness. In *Proc. Seventh Colloquium of Automata, Languages, and Programming*, July 1980.

[19] Peter D. Mosses. Abstract semantic algebras! In *Proc. IFIP TC2 Working Conference on Formal Description of Programming Concepts II (Garmisch-Partenkirchen, 1982).* North-Holland, 1983.

[20] Peter D. Mosses. A basic abstract semantic algebra. In *Proc. Int. Symp. on Semantics of Data Types (Sophia-Antipolis).* Springer-Verlag (*LNCS* 173), 1984.

[21] Peter D. Mosses. Unified algebras and action semantics. In *Proc. STACS'89.* Springer-Verlag, 1989.

[22] Peter D. Mosses. Unified algebras and institutions. In *LICS'89, Fourth Annual Symposium on Logic in Computer Science*, 1989.

[23] Peter D. Mosses. Unified algebras and modules. In *Sixteenth Symposium on Principles of Programming Languages.* ACM Press, January 1989.

[24] Peter D. Mosses. An introduction to action semantics. Technical Report DAIMI IR-102, Computer Science Department, Aarhus University, July 1991. Lecture Notes for the Marktoberdorf'91 Summer School.

[25] Peter D. Mosses. *Action Semantics.* Cambridge University Press, 1992. To appear, in the series *Tracts in Theoretical Computer Science.*

[26] Peter D. Mosses and David A. Watt. The use of action semantics. In *Proc. IFIP TC2 Working Conference on Formal Description of Programming Concepts III (Gl. Avernæs, 1986).* North-Holland, 1987.

[27] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56, 1988.

[28] Hewlett Packard. Precision architecture and instruction. Technical Report 09740–90014, June 1987.

[29] Jens Palsberg. *Provably Correct Compiler Generation.* PhD thesis, Computer Science Department, Aarhus University, 1992. Forthcoming.

[30] Jens Palsberg. A provably correct compiler generator. In *Proc. ESOP'92, European Symposium on Programming*, 1992.

[31] Lawrence Paulson. A semantics-directed compiler generator. In *Ninth Symposium on Principles of Programming Languages*, pages 224–233. ACM Press, January 1982.

[32] Uwe F. Pleban. Compiler prototyping using formal semantics. In *Proc. ACM SIG-PLAN'84 Symposium on Compiler Construction*, pages 94–105. Sigplan Notices, 1984.

[33] Uwe F. Pleban and Peter Lee. On the use of LISP in implementing denotational semantics. In *Proc. ACM Conference on LISP and Functional Programming*, August 1986.

[34] Uwe F. Pleban and Peter Lee. High-level semantics, an integrated approach to programming language semantics and the specification of implementations. In *Proc. Mathmatical Foundations of Programming Language Semantics*, April 1987.

[35] Uwe F. Pleban and Peter Lee. An automatically generated, realistic compiler for an imperative programming language. In *Proc. SIG-PLAN'88 Conference on Programming Language Design and Implementation*, June 1988.

[36] Wolfgang Polak. *Compiler Specification and Verification.* Springer-Verlag (*LNCS* 213), 1981.

[37] David A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[38] Uwe Schmidt and Reinhard Völler. A multi-language compiler system with automatically generated codegenerators. In *Proc. ACM SIG-PLAN'84 Symposium on Compiler Construction.* Sigplan Notices, 1984.

[39] Uwe Schmidt and Reinhard Völler. Experience with VDM in Norsk Data. In *VDM'87. VDM—A Formal Method at Work.* Springer-Verlag (*LNCS* 252), March 1987.

[40] William Stallings. *Reduced Instruction Set Computers.* IEEE Computer Society Press, 1986.

[41] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[42] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. More on advice on structuring compilers and proving them correct. *Theoretical Computer Science*, 15:223–249, 1981.

[43] Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly, 1991.

[44] Mitchell Wand. A semantic prototyping system. In *Proc. ACM SIGPLAN'84 Symposium on Compiler Construction*, pages 213–221. Sigplan Notices, 1984.

[45] David Watt. *Programming Language Syntax and Semantics.* Prentice-Hall, 1991.

[46] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5:493–518, 1989.