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

Jens Palsberg
palsberg@dsi.aau.dk

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

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.

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 correctness. 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 [41] 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 [30], except the correctness proof. For an overview of our approach to correctness, see [31].

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 language designer's workbench.

Common to all of the approaches are that they choose a specific target language [33]. 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.

The traditional approach to compiler generation is based on denotational semantics [38]. Examples of existing compiler generators based on this idea include Moses's Semantics Implementation System (SIS) [17], Paulson's Semantics Processor (PSP) [32, 33], and Wand's Semantic Prototyping System (SPS) [45].

Denotational semantics has achieved much popularity as a vehicle for theoretical studies, but it is also recognized as not being flexible or readable, see for example the discussions by Moses [19], and Pleban and Lee [35]. 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 classical 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 Völker [39, 40], the compiler generator of Kelsey and Hudak [10], and the Mess system of Pleban and Lee [34, 12, 36, 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, Fortran, and Cobol. The specification language, called CAT, is a simplification of the union of all their syntactic constructs. This makes the approach less general than the classical ones, in that it is biased towards a specific style of architecture.

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 to defining languages is high-level semantics. High-level semantics is compositional, but it does not
Figure 2: Existing Compiler Generators.

<table>
<thead>
<tr>
<th>Designer of the system</th>
<th>Specification language</th>
<th>Quality of generated compilers</th>
<th>Correctness Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moses</td>
<td>Denotational Semantics</td>
<td>Poor</td>
<td>No</td>
</tr>
<tr>
<td>Paulson</td>
<td>Denotational Semantics</td>
<td>Poor</td>
<td>No</td>
</tr>
<tr>
<td>Wand</td>
<td>Denotational Semantics</td>
<td>Poor</td>
<td>No</td>
</tr>
<tr>
<td>Schmidt and Völler</td>
<td>Amalgamation of five languages</td>
<td>Good</td>
<td>No</td>
</tr>
<tr>
<td>Kelsey and Hudak</td>
<td>Lambda notation with implicit store, etc.</td>
<td>Good</td>
<td>No</td>
</tr>
<tr>
<td>Pleban and Lee</td>
<td>High-level semantics</td>
<td>Good</td>
<td>No</td>
</tr>
<tr>
<td>Gomard and Jones</td>
<td>Denotational Semantics</td>
<td>Poor</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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, 42, 37, 27], algebraic variations hereof [3, 16, 43, 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 Moses, 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 Moses [19, 20, 21, 24, 25] and Watt [26, 46]. It is intended to allow useful semantic descriptions of realistic programming 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 an excerpt of a description of a subset of Ada, called Mini-Ada, featuring static typing, constants, variables, one-dimensional array-types, functions and procedures with in and out (reference) parameters, various control structures, and the usual expressions. The full description of Mini-Ada is approximately four times longer and is omitted here due to space constraints. It may be found in [29]. The Mini-Ada specification is a subset of one given by Moses 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

The Cantor system has the structure shown in figure 3. In practice, a session with Cantor looks as fol-
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 are 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].

### Table: Pseudo SPARC vs. Real SPARC

<table>
<thead>
<tr>
<th>Pseudo SPARC</th>
<th>Real SPARC</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>sub %g0, %g0, %g0</td>
</tr>
<tr>
<td>jump Z</td>
<td>jmpl %r8, %g0</td>
</tr>
<tr>
<td>branchequal Z</td>
<td>be Z</td>
</tr>
<tr>
<td>branchlessthan Z</td>
<td>jmpl %r8+8, %g0</td>
</tr>
<tr>
<td>call return</td>
<td>jmpl global, %r8</td>
</tr>
<tr>
<td>store R1 in R2 Z P</td>
<td>st R1, R2+Z+P</td>
</tr>
<tr>
<td>load R1 Z P into R2</td>
<td>ld R1+Z+P, R2</td>
</tr>
<tr>
<td>store registers</td>
<td>save</td>
</tr>
<tr>
<td>load registers</td>
<td>restore</td>
</tr>
<tr>
<td>move R1 to R</td>
<td>or %g0, R1, R</td>
</tr>
<tr>
<td>move sum R R1 to R'</td>
<td>add R, R, R'</td>
</tr>
<tr>
<td>move difference R R1 to R'</td>
<td>sub R, R, R'</td>
</tr>
<tr>
<td>compare R with R1</td>
<td>subcc R, R1, %g0</td>
</tr>
</tbody>
</table>

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.

(2) D h d = (n:natural, truth-value-type);
(3) l" = sum(l',e-size);
(4) evaluate D h a f d l sum(l" ,6) =
  (p:program, r:general-register)
⇒ perform [[ "check" D:Dependent ]] h a f d
  u_a u_f l n u_f 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_a 0 l_n,
    putcommit sum(l",6) 0,
    finalize sum(l",8) u_f 2 l_f )

The first definition calculates the size of the code generated by the second 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 Moses
[23, 21, 22]. This includes the semantics of action not-
ation (13 pages), the semantics of the machine lan-
guage (6 pages), the compiler (36 pages), and various
auxiliary notation (14 pages). The correctness state-
ment, 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 au-
tomatically check a compiler correctness proof are
reported by Young [47] and Joyce [9, 8]. For now, how-
ever, we leave the automatic checking of the Cantor
correctness proof to future work and turn to a perfor-
mance evaluation.

4 Performance Evaluation

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

• Generating the Mini-Ada compiler takes 9 sec-
onds.

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 al-

erows 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.

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
bubble: Bubblesorts a number of integers (50 lines).
sieve: Performs the sieve of Eratosthenes 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.

<table>
<thead>
<tr>
<th>No. of Pseudo SPARC instructions generated:</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble: 16697</td>
</tr>
<tr>
<td>sieve: 12096</td>
</tr>
</tbody>
</table>

Figure 6: Object code size.

SPARC, then the size is approximately doubled. A slightly worse blow-up is obtained when compiling to the HP Precision Architecture.

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.

<table>
<thead>
<tr>
<th>bubble</th>
<th>C</th>
<th>C²PF</th>
<th>Mini-Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>2.2</td>
<td>542</td>
</tr>
<tr>
<td>sieve</td>
<td>1.2</td>
<td>2.1</td>
<td>377</td>
</tr>
<tr>
<td>euclid</td>
<td>1.1</td>
<td>1.6</td>
<td>136</td>
</tr>
<tr>
<td>fib</td>
<td>1.1</td>
<td>1.7</td>
<td>210</td>
</tr>
</tbody>
</table>

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.

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. 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 Ørbaek for implementing the Cantor system.

Appendix A: Action Notation

**grammar**:

<table>
<thead>
<tr>
<th>Act</th>
<th>Unf</th>
<th>Tuple</th>
<th>Dep</th>
</tr>
</thead>
<tbody>
<tr>
<td>complete</td>
<td>[ Act Infux Unf ]</td>
<td>[ Unf &quot;or&quot; Act ]</td>
<td>&quot;unfold&quot;.</td>
</tr>
<tr>
<td>escape</td>
<td>[ &quot;(&quot; Dep &quot;)&quot; ] [ Tuple &quot;]&quot; ]</td>
<td>&quot;them&quot;.</td>
<td></td>
</tr>
<tr>
<td>fail</td>
<td>[ &quot;true&quot; ] [ &quot;false&quot; ] [ natural ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td>[ &quot;empty-list&quot; ] [ &quot;&amp;&quot; ] [ [ Type &quot;]&quot; ] [ &quot;list&quot; ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>diverge</td>
<td>[ &quot;closure&quot; ] [ &quot;abstraction&quot; ] [ &quot;of&quot; ] [ Act ] [ &quot;&amp;&quot; ] [ [ &quot;perhaps&quot; ] [ &quot;using&quot; ] [ Data ] [ &quot;]&quot; ] [ &quot;act&quot; ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ Unary Dep ]</td>
<td>[ Dep &quot;is&quot; Dep ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>[ Dep ] [ [ Dep &quot;,&quot; ] Dep ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ [ Dep ] ] [ &quot;it&quot; ]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ &quot;the&quot; ] [ &quot;given&quot; ] [ Datum &quot;&amp;&quot; ] [ natural ]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ &quot;true&quot; ] [ Datum &quot;bound&quot; ] [ &quot;to&quot; ] [ token ]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ &quot;the&quot; ] [ Datum &quot;stored&quot; ] [ &quot;in&quot; ] [ Dep ]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Infix</td>
<td>[ &quot;not&quot; ] [ [ Infux ] [ Infux ] ]</td>
<td>&quot;false&quot;</td>
<td></td>
</tr>
<tr>
<td>Unary</td>
<td>[ &quot;and&quot; ] [ &quot;then&quot; ]</td>
<td>[ &quot;then&quot; ]</td>
<td>[ &quot;before&quot; ]</td>
</tr>
<tr>
<td>Binary</td>
<td>[ [ Dep ] ] [ [ Dep ] ]</td>
<td>[ [ Dep ] ] [ [ Dep ] ] [ [ Dep ] ]</td>
<td>&quot;sum&quot;</td>
</tr>
<tr>
<td>Datum</td>
<td>[ [ Datum ] [ Datum ] ]</td>
<td>&quot;datum&quot;</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>[ Datum ] [ Datum ] [ Datum ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ &quot;empty-list&quot; ] [ &quot;&amp;&quot; ] [ [ Type ] [ &quot;list&quot; ] ]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ &quot;rate&quot; ] [ &quot;cell&quot; ] [ &quot;integer&quot; ]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ [ Type ] [ &quot;&amp;&quot; ] [ Datum ] ]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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.

Appendix B:
Mini-Ada Action Semantics (excerpt)

**B.1 Abstract Syntax**

**grammar:**

```
Program = [ Declarations Identifier ] .
Declarations = [ Declarations Declarations ] | ... 
[ identifier := "constant" ] := Expression ; |
[ "procedure" Identifier (" Formals ") ] is Block ; .
Formals = ...
Statements = [ Statements Statements ] | ... 
[ "null" ; ] |
[ Name := " Expression ;" ] |
[ "if" Expression "then" Statements 
  "else" Statements "end" "if" ; ; ] |
[ "while" Expression 
  "loop" Statements "end" "loop" ; ; ] |
[ "exit" ; ] |
[ "declare" Declarations  
  "begin" Statements "end" ; ; ] |
[ Identifier (" Names ") ; ; ] |
[ "return" ; ] .
Block = ...
[ Declarations "begin" Statements "end" ].
Names = ...
Name = ...
Expression = ...
Identifier = token .
```

**B.2 Semantic Entities**

```
item = truth-value | integer .
parameter-less-procedure = abstraction .
parameterized-procedure = abstraction .
escape-reason = [integer] list .
exit = list of 0 .
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 .
  parameterized-procedure-closure . ::
act — dependent datum .
parameterized-procedure-closure A:act =  
```

124
B.3 Semantic Functions

- run .:: Program . act .
  run D:Declarations I:Identifier .
  | furthermore elaborate D .
  | hence .
  | enact application .
  | (the parameter-less-procedure bound to id I) .
  | to .

- elaborate .:: Declarations . act .
  elaborate D1:Declarations D2:Declarations .
  | elaborate D1 before elaborate D2 .
  elaborate I:Identifier ":=" E:Expression .
  | evaluate E then bind id I to it .

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

- execute .:: Statements . act .
  execute S1:Statements S2:Statements .
  | execute S1 and then execute S2 .
  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" S1:Statements .
  | check (is true) and then execute S1 .
  | or .
  | check (is false) and then execute S2 .
  execute "while" E:Expression "loop" S:Statements .
  | unfolding .
  | evaluate E then .
  | check (is true) and then .
  | execute S and then unfold .
  | or check (is false) .
  trap check there-is-given-an-exit .
  | or .
  | check there-is-given-a-return and then escape .
  execute "exit" ";" . give exit then escape .
  execute "declare" D:Declarations .
  | furthermore elaborate D hence .
  | execute S .
  execute I:Identifier "(" N:Names ")" ";" .
  | give the parameterized-procedure bound to id I .
  | and then multi-investigate N .
  | hence .
  | enact application .
  | the given abstraction #1 to the given list #2 .
  execute "return" ";" .
  | give procedure-return then escape .

- execute-block .:: Block . act .
  execute-block D:Declarations .
  | furthermore elaborate D hence .
  | execute S .
  multi-investigate .:: Names . act .
  | investigate .:: Name . act .
  | evaluate .:: Expression . act .
  | id .:: Identifier . token .
  id k:token = k .

References


