A Provably Correct Compiler Generator

Jens Palsberg
palsberg@daimi.aau.dk

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

Abstract

We have designed, implemented, and proved the correctness of a compiler generator that accepts action semantic descriptions of imperative programming languages. The generated compilers emit absolute code for an abstract RISC machine language that currently is assembled into code for the SPARC and the HP Precision Architecture. Our machine language needs no run-time type-checking and is thus more realistic than those considered in previous compiler proofs. We use solely algebraic specifications; proofs are given in the initial model.

1 Introduction

The previous approaches to proving correctness of compilers for non-trivial languages all use target code with run-time type-checking. The following semantic rule is typical for these target languages:

$$(FIRST : C, \langle v_1, v_2 \rangle : S) \rightarrow (C, v_1 : S)$$

The rule describes the semantics of an instruction that extracts the first component of the top-element of the stack, *provided* that the top-element is a pair. If not, then it is implicit that the executor of the target language halts the execution. Hence, the executor has to do run-time type-checking.

Run-time type-checking imposes an unwelcome penalty on execution time because more work has to be done by the executor of the target language. It may be argued, though, that the executor can rely on the source language being statically type-checked, and thus avoid the run-time type-checks. This implies an unwelcome *coupling* of the source and target languages, however, which prevents the target language from being an independent product, for general use.

This paper addresses the use of independent, realistic target languages without type information in the semantics. The paper also concerns the possibility of proving correctness of a compiler *generator*, thus making correctness proof a once-and-for-all effort.

We have overcome these problems. We have designed, implemented, and proved the correctness of a compiler generator, called Cantor, that accepts action semantic descriptions of programming languages. The generated compilers emit absolute code for an abstract RISC [57] machine language without run-time type-checking. The considered subset of action notation, see appendix A, is suitable for describing imperative programming languages featuring:

- Complicated control flow;
- Block structure;
- Non-recursive abstractions, such as procedures and functions; and
- Static typing.

For an example of a language description that has been processed by Cantor, see appendix B. The abstract RISC machine language can easily be expanded into code for existing RISC processors. Currently, implementations exist for the SPARC [25] and the HP Precision Architecture [42].

The technique needed for managing without run-time type-checking in the target language is the following:

- Define the relationships between semantic values in the source and target languages with respect to *both* a type and a machine state.

Thus, we define an operation which given a target value V, a machine state M, and a type T will yield the *sort* of source values which have type T and are represented by V and M. Here, "sort" can be thought of as "set". For example, an integer can represent a value of type truth-value-list by pointing to a heap where the list components are represented. In this case, our operation will yield a sort containing precisely that truth-value-list, when given the integer, the type "truth-value-list", and the heap.

In contrast, for example Nielson and Nielson [40] does *not* involve the machine state when relating semantic values. Instead, they require target values to be "self-contained". Hence, they need to have several types of target values and a target machine that does run-time type checking.

With our approach we can make do with just *one* type of target values, namely integer, thus avoiding run-time type-checking and getting close to the 32-bit words used in the SPARC. Note that we do *not* insert type tags in the run-time representations of source values; *no* type information is present at run-time.

The relationship between semantic values allows the proof of a lemma expressing "code well-behavedness" which is essential when reasoning about executions of compiled code. The required type information is useful during compilation, too; it is collected by the compiler in a separate pass before the code generation. This pass also collects the information needed for generating absolute, rather than relative, code.

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.

The specification and proof of correctness of the Cantor system is an experiment in using the framework of unified algebras, developed by Mosses [35, 33, 34]. Unified algebras allows the algebraic specification of both abstract data types and operational semantics in a way such that initial models are guaranteed to exist, except when axioms contradict constraints, in which case *no* models of the specification exist. We have demonstrated that also a non-trivial compiler can be elegantly specified using unified algebras. In comparison with structural operational semantics and natural semantics, we replace inference rules by Horn clauses. The notational difference is minor, and only superficial differences appear in the proofs of theorems about unified specifications. Where Despeyroux [10] could prove lemmas by induction in the length of inference, we instead adopt an axiomatization of Horn logic and prove lemmas by induction in the number of occurrences of "modus ponens" in the proof in the initial model.

This paper gives an overview of the author's forthcoming PhD thesis [44]. Most definitions and proofs are omitted. For an overview of our experiments with generating a compiler for a subset of Ada, see [43].

In the following section we examine the major previous approaches to compiler generation and compiler correctness proofs. In section 3 we outline the structure of the Cantor system, including the abstract RISC machine language and the action compiler, and we give some performance measures. In section 4 we state the correctness theorem, and finally in section 5 we survey our approach to proving correctness in the absence of run-time type-checking in the target language. We also discuss why we do not treat recursion.

The reader is assumed to be familiar with algebraic specification [12], compilation of block structured languages [64], and the notion of a RISC architecture [57].

2 Previous Work

2.1 Compiler Generation

The problem of compiler generation is usually approached by choosing a particular definition of a specific target language [46]. The task is then to write and prove the correctness of a compiler for a notation for defining source languages. Such a compiler can then be composed with a language definition to yield a correct compiler for the language, see figure 1. Compiler generators that

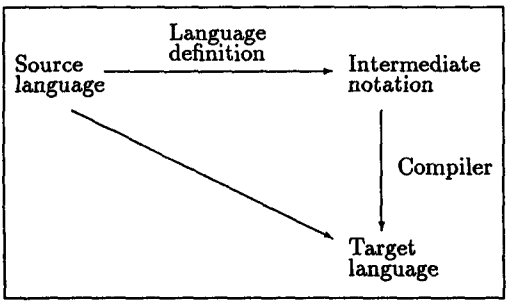

Figure 1: Semantics-directed compiler generation.

operate in this way are often called *semantics-directed* compiler generators. The Cantor system described in this paper is an example of a semantics-directed compiler generator. It accepts language definitions written in action notation, and it outputs compilers that emit code in an abstract RISC machine language.

The traditional approach to compiler generation is based on *denotational semantics* [53]. Examples of existing compiler generators based on this idea include Mosses' Semantics Implementation System (SIS) [29], Paulson's Semantics Processor (PSP) [45, 46], and Wand's Semantic Prototyping System (SPS) [62]. In SIS, the lambda expressions are executed by a direct implementation of beta-reduction; in PSP and SPS they are compiled into SECD and Scheme code, respectively. There are no considerations of the possible correctness of either the implementation of beta-reduction, the translations to SECD or Scheme code, or the implementation of SECD or Scheme. The target programs produced by these systems have been reported to run at least three orders of magnitude slower than corresponding target programs produced by handwritten compilers [22].

After these systems were built, several translations of lambda notation into other abstract machines have been proved correct. Notable instances are the categorical abstract machine [8] and the abstract machines that can be derived systematically from an operational semantics of lambda notation, using Hannan's method [16, 14, 15]. It remains to be demonstrated, however, if a compiler which incorporates one of them will be more efficient than the classical systems. Also, the correctness of implementations of these abstract machines has not been considered.

It appears that the poor performance characteristics of the classical compiler generators do not simply stem from inefficient implementations of lambda notation. Mosses observed that denotational semantics intertwine model details with the semantic description, thus blurring the underlying conceptual analysis [31]. Pleban and Lee further observed that not only a human reader but also an automatic compiler generator will have difficulty in recovering the underlying analysis [48]. Attempts to recover useful information from lambda expressions include Schmidt's work on detecting so-called single-threaded store arguments and stack single-threaded environment arguments [52, 54], and the binding-time analysis of Nielson and Nielson [41]. Despite that, it seems unlikely that the performance characteristics of compiler generators based on denotational semantics soon will be improved beyond that of existing such systems.

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 [55, 56], the compiler generator of Kelsey and Hudak [21], and the Mess system of Pleban and Lee [47, 23, 49, 22]. These approaches are based on rather ad hoc notations for defining languages, and they lack correctness proofs, like the classical systems. They indicate, however, that better performance of the produced compiler is obtained when:

- Some model details are omitted from a language definition; and
- The notation for defining languages is biased towards "compilable languages".

A radically different approach to compiler generation is taken by Dam and Jensen [9]. They consider the use of natural semantics [20] (which they call "relational semantics") as the basis of a compiler generator. They devise an algorithm for transforming a natural semantic definition into a compiling specification. The algorithm requires a language definition to satisfy some conditions; it is sufficiently general to apply to a language of while-programs, but has not been implemented. The generated compilers emit code for a stack machine; the correctness of these compilers has been sketched, whereas the implementation of the stack machine is not considered.

Finally, compiler generation can be obtained by self-application of a partial evaluator. The Ceres system of Tofte [60] is an early example of this, demonstrating that even compiler generators can be automatically generated. Ceres uses a language of flowcharts with an implicit state as the notation for defining source languages. Another notable partial evaluator is the Similix of Bondorf and Danvy [5, 6] which treats a subset of Scheme. Gomard and Jones implemented a self-applicable partial evaluator, called mix, for an untyped lambda notation [13]. It has been used to generate a compiler for a language of while-programs. The generated compiler emits programs in lambda notation. The correctness of this compiler generator has been proved; it remains to be seen, however, if the partial evaluation approach will lead to the generation of compilers for conventional machine architectures.

The lack of correctness proofs for the realistic compiler generators limits the confidence we can have in a generated compiler. Let us therefore examine the major previous approaches to compiler correctness proofs.

2.2 Compiler Correctness Proofs

The traditional approach to proving compiler correctness is based on denotational semantics [24, 26, 58, 51, 39] or algebraic variations hereof [7, 28, 59, 3, 30]. The correctness statement can be pictured as a commuting diagram, see figure 2.

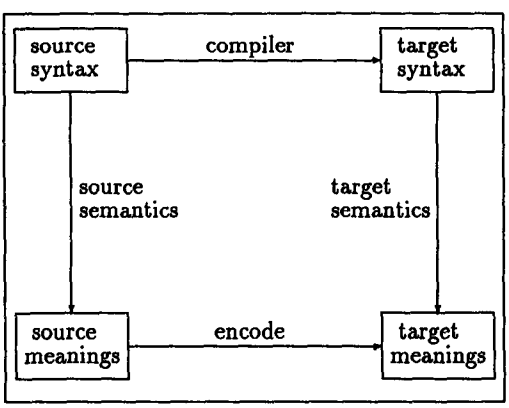

Figure 2: Compiler correctness.

It has been demonstrated that complete proofs of compiler correctness can be automatically checked. Two significant instances are Young's [65] work, using the Boyer-Moore theorem prover, and Joyce's [19, 18] work using the HOL system. In both cases, the target code of the translation is a non-idealized machine-level architecture whose implementation has been verified with respect to a low level of the computer, see for example [17, 27]. The verification of both architectures has even been automatically checked. These examples of systems verification [4] are important: they minimize the amount of distrust one need have to such a verified system. Of course, one can still suspect errors in the implementation of the gate-level of the computer, or in the implementation of the theorem prover, but many other sources of errors have been eliminated.

The use of denotational semantics renders difficult the specification of languages with nondeterminism and parallelism. Such features can be specified easily, however, by adopting the framework of structural operational semantics [50]. For a survey of recent work on proving the correctness of compilers for such languages, see the paper by Gammelgaard and Nielson [11], which also contains a detailed account of the approach taken in the ProCoS project, where the source language considered is Occam2.

In a special form of structural operational semantics, called natural semantics [20], one considers only steps from configurations to *final* states. When both the source and target languages have a natural semantics, then there is hope for proving the correctness of a compiler using the proof technique of Despeyroux [10]. As with the proof techniques used when dealing with denotational semantics, Despeyroux's technique amounts to giving a proof by induction on the length of a computation. The correctness statement is different, though. Instead of proving that a diagram commutes, she proves the validity of two properties, which informally can be stated as follows:

- **Completeness:** if the source program terminates, then so does the target program, and with the same result; and
- **Soundness:** if the target program terminates, then so does the source program, and with the same result.

Despeyroux proves the correctness statement by induction in the length of the proofs of the assumptions of these properties. A central lemma

states that the code for an expression behaves in a disciplined way. We call this property "code well-behavedness". We will use a variation of Despeyroux's technique, adapted to the framework of unified algebras, see later.

A major deficiency of all the previous approaches to compiler correctness, except that of Joyce [19, 18], is their using a target language that performs *run-time* type-checking, as explained above. Joyce considers only a language of while-programs, and it is not clear how to generalize his approach.

Our concern can be sloganized as follows:

- If "well-typed programs don't go wrong", then it should be possible to generate correct code for an independent, realistic machine language that does not perform run-time type-checking.

The Cantor system is based on the use of such a machine language.

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 [31, 32, 33, 37, 36] and Watt [38, 63]. 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 [36]. The syntax of this subset is given in appendix A together with a brief overview of the principles behind action semantics. Appendix B presents a complete description of a toy programming language. (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 [36] for a full presentation of action semantics.)

The central part of the Cantor system is a compiler from action notation to an abstract RISC machine language. This section presents both the machine language and the compiler, and it states some performance measurements of the Cantor system.

All specifications in this paper, including those of syntax, are given in Mosses' meta-notation for unified algebras [36].

3.1 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 the following machine state:

```
sparc-state =
    (program, program-counter, was-zero,
    was-negative, globals, windows, memory) .
```

'program' is a mapping from linenumbers to instructions. 'program-counter' is a linenumber, and 'was-zero' and 'was-negative' are status-bits (truth-values). 'globals' models the global registers, and 'windows' models a non-overlapping version of the SPARC register-windows. Finally, 'memory' models six separate "pages" of the main memory, as a mapping from page-identifications to pages. A page is a mapping from addresses (natural numbers) to integers. For example, one of the pages is used as a stack, another as a heap.

The only data manipulated by this language are integers. This means that it is impossible to see from a given data value if it should be thought of as a pointer to an instruction in the program, as an address in the memory, or as modeling a truth-value, an integer, etc.

The uniformity of the data values makes the Pseudo SPARC language more realistic than those considered in previous compiler proofs. It contains two major 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".

These idealizations simplify the correctness proof considerably, without removing any of the difficulties that we address.

Pseudo SPARC	Real SPARC
skip	`sub %g0, %g0, %g0`
jump *Z*	`jmpl Z, %g0`
branchequal *Z* branchlessthan *Z*	`be Z` `bneg Z`
call return	`jmpl global, %r8` `jmpl %r8+8, %g0`
store *R1* in *R2 Z P* load *R1 Z P* into *R2*	`st R1, R2+Z+P` `ld R1+Z+P, R2`
storeregisters loadregisters	`save` `restore`

move *RI* to *R*
move sum *R RI* to *R'*
move difference *R RI* to *R'*
	`or %g0, RI, R`
	`add R, RI, R'`
	`sub R, RI, R'`

compare *R* with *RI*	`subcc R, RI, %g0`

Figure 3: The Pseudo SPARC machine language.

Figure 3 shows the 14 Pseudo SPARC instructions and how they (approximately) can be expanded to real SPARC instructions. In practice, the expansion has to take care of fitting instructions using large integers into several real SPARC instructions. It also has to insert additional "nop" instructions into so-called "delay slots". Pseudo SPARC instructions can also be expanded to instructions for the HP Precision Architecture, though with a little more difficulty.

The function that models one step of computation is defined as follows:

```
step _ :: sparc-state → sparc-state (total) .

step m = next
    ((program of m) at
     (program-counter of m) default skip) m .
```

'step _' models the loading of the current instruction, followed by its execution. The operation 'next _ _ ' is defined in the following style (we give only a single example):

```
next _ _ ::
    instruction, sparc-state → sparc-state (total) .

next call (p, pc, cz, cn, g, w, q) =
    (p, g at global default 0, cz, cn, g,
    update w (map of return-address to pc), q) .
```

Here, 'global' is one of the global registers, and 'return-address' is a user-inaccessible register in the register-window. The use of 'default' models that all registers and memory addresses are initialized to 0 before execution starts. Likewise, the program area contains 'skip' instructions everywhere before the program is loaded.

Note that 'step _' and 'next _ _' are total functions. This emphasizes that computation continues infinitely, once started. For example, the 'call' instruction will be executed even though the global register contained a value that we thought of as a truth-value! It also means that we have avoided alignment problems, etc., so that a typical run-time error such as "bus error" will not occur. This is accomplished by having a word- rather than byte-oriented definition of the Pseudo SPARC machine.

3.2 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 (we cheat a little bit here, compared to [44], to improve the readability):

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

perform _ _ _ _ _ _ _ _ _ _ _ _ ::
    Act, data-type, 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, e.g., 'complete', 'escape', 'fail', 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,

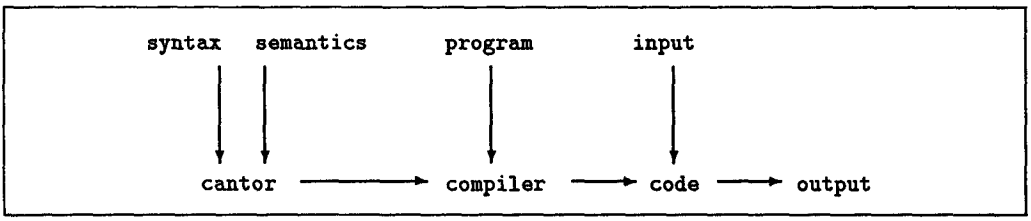

Figure 4: The Cantor system.

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.

The function 'a-count' is defined as a forwards abstract interpretation, computing with types of tuples of data ('data-type'), types of bindings ('symbol-table'), and code sizes ('natural'). The first 'truth-value' component tells if the action being analyzed has a chance of completing. If it does, then the following 'data-type' component tells the type of the tuple of data that will produced. The next two components give similar information about escaping.

The function 'perform' takes as arguments the 'data-type' and 'symbol-table' that are also supplied to 'a-count'. In addition, it takes a 'general-register' which at run-time will contain a pointer to a representation of the tuples of data that will be received when executing then code. The set 'frozen' contains those registers that the code to be produced must not modify, and the three 'cleanup' values are natural numbers that indicate how much to pop from the stack, should the performance complete, escape, or fail.

The calculation of whether an action can complete or not, and whether it can escape or not, are examples of the compile time analyses that are built into the compiler. They are used to generate better code, and they are fully integrated in the proof of correctness, see later.

3.3 Performance Evaluation

The Cantor system has the structure shown in figure 4. 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 [61], 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.

The HypoPL language, defined in appendix B, is taken from Lee's book on realistic compiler generation [22], with the difference that we treat nesting of procedures in its full generality but do not allow recursion. (For a discussion of why recursion is problematic, see later.)

- Generating a compiler for HypoPL takes 3 seconds.

We have used this compiler to translate Lee's bubblesort program (50 lines).

- Compile time: 486 seconds;
- Object code size: 114688 bytes; and
- Object code execution time (for sorting 10 integers): 0.1 seconds.

These figures indicate that the system is rather tedious to work with in practice. Additional experiments, see [43], have shown that the code runs at least two orders of magnitude slower than a corresponding target program produced by the C compiler (without optimization). This is somewhat disappointing but 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 [22]. Inspection of the code emitted by Cantor-generated compilers reveals that the inefficiency mainly stems from three sources:

- Lack of compile time constant propagation;
- Poor register allocation; and
- Naive representation of bindings, closures, and lists.

Improving the action compiler to avoid this inefficiency would significantly complicate the correctness theorem, which we consider next.

4 The Correctness Theorem

To give an overview of the correctness theorem, we will introduce a bit of notation, as follows (we cheat a little bit again, compared to [44], to improve the readability):

run _ _ :: Act, [integer] list → state .

sparc-run _ _ ::
    program, natural, page → sparc-state .

compile _ :: Act →
    (program, truth-value, data-type,
    truth-value, data-type,
    general-register, general-register) .

abstract _ _ _ _ _ _ _ ::
    sparc-state, truth-value, data-type,
    truth-value, data-type,
    general-register, general-register → state .

i-abs _ _ :: natural, page → [integer] list .

(1) a-count A () (list of empty-list) =
    $(n, z_n, h_n, z_e, h_e,$ empty-list) ;

(2) perform A () (reg 0) empty-set
    (list of empty-list) 0 0 0 0 n n n =
    (p, a_n, a_e)

⇒ compile $A = (p, z_n, h_n, z_e, h_e, a_n, a_e)$.

We have only given the definition of 'compile _', in terms of 'a-count' and 'perform'. The operations have the following informal meaning:

1. The operation 'run A il' specifies the performance of an action A which is given the empty tuple of data, no bindings, an empty-storage, an empty output-file, and the input-file il (an integer-list). If the performance terminates, then that will result in a final state ('state') which can be either completed, escaped, or failed.
2. The operation 'sparc-run p n se' specifies loading the program p into the program area, and then taking n steps starting in line 0. It also records if the execution at any point "jumps outside the code". The memory, registers, status bits, and output file are initialized appropriately, the input file is initialized to se. 'sparc-run' is defined in terms of 'step', described above.
3. The operation 'compile A' translates the action A into a machine language program p and it also gives type information about what will be produced when performing A. The program p will start in line 0.
4. The operation 'abstract m_p z_n h_n z_e h_e a_n a_e' will give a *sort* of all those states (from the action-level) that are represented by the sparc-state m_p, and that have the type expressed by the following four arguments. The last two arguments are those registers which will contain pointers to the representations of the data produced, should the action complete or escape.
5. The operation 'i-abs n se' will give the input-file ('[integer] list') which is represented by the natural number n and the page se.

The use of both type information and a machine-state in the definition of 'abstract' makes it possible to make do without type information in the semantics of Pseudo SPARC.

None of the above five operations are total. The performance of an action may diverge; the execution of a machine program may "jump outside the code"; the compilation of an action may find a type error; the machine state may represent no state at all from the action-level; and the page for input-files may contain something without the right format.

The meta-notation for unified algebras makes it particularly easy to specify such partial operations. This is because it supports a *unified* treatment of sorts and individuals: an individual is treated as a special case of a sort. Thus operations can be applied to sorts as well as individuals. A vacuous sort represents the lack of an individual, in particular the 'undefined' result of a partial operation. For example, if the performance of the action A with input-file il terminates, then 'run A il' will be an individual,

otherwise it will be a vacuous sort. We need not specify that such sorts are vacuous; if it does not follow from the specification that they contain an individual, then they will automatically be vacuous.

The operations 'run', 'sparc-run', 'compile', and 'i-abs' will all yield either an individual or a vacuous sort. In contrast, 'abstract' may yield a sort containing *several* individuals, and it may also yield a vacuous sort. The possibility of yielding a sort containing several individuals is needed when abstracting with respect to a closure type. This is because if two actions differs only in the naming of tokens (they are equal with respect to "alpha-conversion"), then the compiled code for them will be identical.

We can now state the correctness theorem. Note that 't :- s' is another syntax for 's : t'. The meaning is that s is an individual contained in t.

Theorem:

(1) compile A:Act =
(p:program z_n:truth-value h_n:data-type
z_e:truth-value h_e:data-type
a_n:general-register a_e:general-register) ;

(2) i-abs (se at 0) se = il:[integer] list

⇒ (1) run A il = m_a:state ⇒
(∃ m_p:sparc-state ∃ n:natural .
sparc-run p n se = m_p ;
abstract m_p z_n h_n z_e h_e a_n a_e :- m_a) ;

(2) sparc-run p n se = m_p:sparc-state ⇒
(∃ m_a:state .
run A il = m_a ;
abstract m_p z_n h_n z_e h_e a_n a_e :- m_a) .

The structure of the theorem resembles the correctness statement of Despeyroux. Informally:

If the action A is compiled into a machine language program p (and some additional type information, etc., is produced), and the input-file il is represented properly in the machine as se, then two properties hold:

1. **Completeness:** If the performance of the action A (with input-file il) terminates in state m_a, then there exists a sparc-state m_p and a number n such that an n-step execution of p will reach m_p, and m_p represents m_a (and the program-counter points to the last line of p).
2. **Soundness:** If an n-step execution of p (with input se) reaches m_p (and the program-counter points to the last line of p), then there exists a state m_a, represented by m_p, such that a performance of A (with input il) will terminate in m_a.

Notice that it is built into the definition of 'sparc-run', and hence the correctness theorem, that the execution of the machine language program never "jumps outside the code".

5 The Proof Technique

A number of lemmas are needed to prove the theorem; here is an overview:

- **Compiler consistency:** These lemmas state that the calculation of code size is correct. They also state that the code is placed consecutively, starting in the desired line.
- **Correctness of analysis:** These lemmas state that the type analysis asserts correct typings, relative to the semantics of actions.
- **Code well-behavedness:** These lemmas state that if the execution of some compiled code at some point reaches "the end of the code", then it used the memory and registers in a disciplined fashion, and, in addition, the machine state will represent an abstract state (with the type given by the compiler).

It is also necessary to prove strengthened versions of completeness and soundness.

Let us now consider how to adopt Despeyroux's proof technique to the framework of unified algebras.

Despeyroux expresses natural semantics in the Gentzen's system style, with axioms and inference rules. In such a system one can make natural deduction, and can then prove lemmas about the system by induction in the length of such deductions. In contrast, the framework of unified algebras provide Horn clauses, and there are no build-in deduction rules. We have replaced inference rules by Horn clauses, so to be able to do deduction, we adopt a standard axiomatization of Horn logic, as follows.

All specifications in the meta-notation for unified algebras can be transformed into a core notation which is outlined in the following. Let Ω

$$\frac{}{(\Omega,\Gamma)\vdash t=t} \qquad \text{(Reflexivity)}$$

$$\frac{(\Omega,\Gamma)\vdash s=t \qquad (\Omega,\Gamma)\vdash t=u}{(\Omega,\Gamma)\vdash s=u} \qquad \text{(Transitivity)}$$

$$\frac{\{(\Omega,\Gamma)\vdash s_i=t_i\}_{i=1}^{n}}{(\Omega,\Gamma)\vdash f(s_1,\ldots,s_n)=f(t_1,\ldots,t_n)} \text{ if } f\in\Sigma \qquad \text{(Functional Congruence)}$$

$$\frac{\{(\Omega,\Gamma)\vdash s_i=t_i\}_{i=1}^{2} \qquad (\Omega,\Gamma)\vdash p(s_1,s_2)}{(\Omega,\Gamma)\vdash p(t_1,t_2)} \text{ if } p\in\{\_\leq\_,\_:\_\} \qquad \text{(Predicative Congruence)}$$

$$\frac{\{(\Omega,\Gamma)\vdash F_i\}_{i=1}^{n}}{(\Omega,\Gamma)\vdash F} \text{ if } (F_1;\ldots;F_n\Rightarrow F)\in\Gamma \qquad \text{(Modus Ponens)}$$

Figure 5: Axiomatization of Horn clause logic.

be a so-called homogeneous first-order signature, that is, a pair $\langle\Sigma,\Pi\rangle$ where Σ is a set of operation symbols and Π is a set of predicate symbols. In the setting of unified algebras, it is required that

$$\Sigma\supseteq\{\text{nothing},\ \_\mid\_,\ \_\&\_\}$$

and

$$\Pi=\{\_=\_,\ \_\leq\_,\ \_:\_\}$$

The value of the constant 'nothing' is a vacuous sort, included in all other sorts. The operation '$\_\mid\_$' is sort union, and '$\_\&\_$' is sort intersection. The predicate '$\_=\_$' asserts equality, '$\_\leq\_$' asserts sort inclusion, and '$T_1:T_2$' asserts that the value of the term T_1 is an individual included in the (sort) value of the term T_2.

Further, let Γ be a set of Horn clauses built up from Ω. Any specification Γ of such Horn clauses will be augmented with some basic Horn clauses, stating for example the reflexivity of '$\_\leq\_$', see [34]. Finally, let F be a formula built up from Ω. We will then write

$$(\Omega,\Gamma)\vdash F$$

(read F is (Ω,Γ)-deducible) if $(\Omega,\Gamma)\vdash F$ can be obtained by finitely many applications of the deduction rules shown in figure 5. A deduction rule consists of a *conclusion* (given beneath the line), none, one, or several *premises* (given above the line), and possibly a *condition* (given at the right-hand side of the line). A deduction rule stands for the statement:

- If all premises are deducible, the condition is satisfied, and F is a formula built up from Ω, then the conclusion $(\Omega,\Gamma)\vdash F$ is deducible.

With these deduction rules, we can do proof by induction in the number of occurrences of "modus ponens" in deductions. Note that a single application of modus ponens corresponds closely to a natural deduction step. This makes our proof strategy close to Despeyroux's. All lemmas proved by induction in the length of deduction are satisfied by the *initial* model of the specification. The key property of an initial model needed here is that it only contain entities that are values of ground terms (it contains "no junk").

We will end this section by explaining why we do not treat recursion, in contrast to Despeyroux. The reason for this is rather subtle; it hinges on the expressiveness of the unified meta-notation.

Full action notation offers self-referential bindings as the means for describing for example recursive procedures. A self-referential binding is a cyclic structure; the run-time representation will obviously also be cyclic. In Despeyroux's paper, such cyclic structures are represented as graphs with self-loops—both in the source and target languages. This allows her to uniquely determine the run-time representation of a self-referential environment.

Compared to Despeyroux, we use a much more low-level target language where values can be placed in more than one place in the memory. This means that not only can one target value represent more than one source value, as in Despeyroux's paper, it is also possible for one source value to be represented by different parts of the memory. In other words, there is no *functional* connection between source and target values; there is only a *relation* stating which source values are represented by a given part of the mem-

ory.

In the case of cyclic structures, the relation between semantic values seems to be impossible to define in the unified meta-notation. This is because the meta-notation only allows the expression of Horn clauses. Evidence for this is found in Amadio and Cardelli's paper on subtyping recursive types [2]. They axiomatize several relationships between cyclic structures, and it seems that a rule of the following non-Horn kind cannot be avoided:

$$(x \; R \; y \Rightarrow \alpha \; R \; \beta) \Rightarrow \mu x.\alpha \; R \; \mu y.\beta$$

Since we want to apply the unified meta-notation exclusively in all specifications, we avoid self-referential bindings. Thus we cannot treat recursion.

6 Conclusion

Our compiler generator is specified and proved correct solely in an algebraic framework. To our knowledge, it is the first time that this has been accomplished.

The generated compilers emit realistic, albeit poor, machine code. Future work includes building in more analyses, for the benefit of the code generator.

The use of action semantics makes the processable specifications easy to read and pleasant to work with. We believe that the Cantor system is a promising first step towards user-friendly and automatic generation of realistic and *correct* compilers.

Acknowledgements. 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, Michael Schwartzbach, and the referees for helpful comments on a draft of the paper. The author also thanks Peter Ørbæk for implementing the Cantor system.

Appendix A: Action Notation

grammar:

```
Act    = "complete" | "escape" | "fail" |
         "commit" | "diverge" | "regive" |
         [ "give" Dep ] | [ "check" Dep ] |
         [ "bind" token "to" Dep ] |
         [ "store" Dep "in" Dep ] |
         [ "allocate" "truth-value" "cell" ] |
         [ "allocate" "integer" "cell" ] |
         [ "batch-send" Dep ] |
         [ "batch-receive" "an" "integer" ] |
         [ "enact" "application" Dep
         "to" Tuple ] |
         [ "indivisibly" Act ] |
         [ "unfolding" Unf ] |
         [ Act Infix Act ] |
         [ [ "furthermore" Act ] "hence" Act ] |
         [ [ "furthermore" Act ] "thence" Act ] .

Unf    = [ Act Infix Unf ] | [ Unf "or" Act ] |
         "unfold" .

Tuple  = "()" | Dep | [ Tuple "," Tuple ] |
         "them" .

Dep    = "true" | "false" | natural |
         [ "empty-list" "&" "[" Type "]" "list" ] |
         [ "closure" "abstraction" "of" Act "&"
         "[" "perhaps" "using" Data "]" "act" ] |
         [ Unary Dep ] |
         [ Binary "(" Dep "," Dep ")" ] |
         [ Dep "is" Dep ] |
         [ Dep [ "is" "less" "than" ] Dep ] |
         [ "component#" Dep "items" Dep ] |
         "it" |
         [ "the" "given" Datum "#" natural ] |
         [ "the" Datum "bound" "to" token ] |
         [ "the" Datum "stored" "in" Dep ] |
         [ "(" Dep ")" ] .

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-value" "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.

Appendix B: HypoPL Action Semantics

B.1 Abstract Syntax

grammar:

```
Program     = [[ "program" Identifier Block ]] .

Declaration = [[ "int" Identifier ]] |
              [[ "bool" Identifier ]] |
              [[ "const" Identifier "=" Integer ]] |
              [[ "array" Identifier "[" Integer "]" ]] |
              [[ "procedure" Identifier
              "(" Identifier ")" Block ]] |
              [[ Declaration ";" Declaration ]] .

Block       = [[ Declaration
              "begin" Statement "end" ]] |
              [[ "begin" Statement "end" ]] .

Statement   = [[ Expression ":=" Expression ]] |
              [[ "write" Expression ]] |
              [[ "read" Expression ]] |
              [[ "if" Expression "then" Statement
              "else" Statement "endif" ]] |
              [[ "while" Expression "do"
              Statement "endwhile" ]] |
              [[ Identifier "(" Expression ")" ]] |
              [[ Statement ";" Statement ]] | "skip" .

Expression  = "true" | "false" | Integer |
              Identifier |
              [[ Identifier "[" Expression "]" ]] |
              [[ Expression Operation Expression ]] |
              [[ "not" Expression ]] .

Operation   = "+" | "-" | "<" | "=" | "and" .

Integer     = natural | [[ "-" natural ]] .

Identifier  = token .
```

B.2 Semantic Entities

B.2.1 Items

introduces: item .

item = truth-value | integer .

B.2.2 Coercion

introduces: coercively _ .

• coercively _ :: act → act .

coercively *A*:act =
  | *A*
  then
  | give the given item #1 or
  | give the item stored in the given cell #1 .

B.3 Semantic Functions

introduces:
run _ , establish _ , activate _ , execute _ ,
evaluate _ , operation-result _ ,
integer-value _ , id _ .

B.3.1 Programs

• run _ :: Program → act .

run [["program" *I*:Identifier *B*:block]] = activate *B* .

B.3.2 Declarations

• establish _ :: Declaration → act .

establish [["int" *I*:Identifier]] =
  allocate integer cell then bind id *I* to it .

establish [["bool" *I*:Identifier]] =
  allocate truth-value cell then bind id *I* to it .

establish [["const" *I*:Identifier "=" *j*:integer]] =
  bind id *I* to integer-value *j* .

```
establish [[ "array" I:Identifier "[" j:integer "]" ]] =
    | give empty-list & [integer cell] list and then
    | give sum(integer-value j, 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
  then
    | bind id I to the given list #1 .
```

```
establish ⟦ "procedure" I1:Identifier
      "(" I2:Identifier ")" B:Block ⟧ =
    bind id I1 to
    closure abstraction of
    | | furthermore
    | | | give the given integer #1 and then
    | | | allocate integer cell
    | | then
    | | | store the given integer #1
    | | |       in the given cell #2
    | | and then
    | | | bind id I2 to the given cell #2
    | thence activate B
    & [perhaps using integer] act .

establish ⟦ D1:Declaration ";" D2:Declaration ⟧ =
    establish D1 before establish D2 .
```

B.3.3 Blocks

```
• activate _ :: Block → act .

activate ⟦ D:Declaration
      "begin" S:Statement "end" ⟧ =
    | furthermore establish D
    hence execute S .

activate ⟦ "begin" S:Statement "end" ⟧ = execute S .
```

B.3.4 Statements

```
• execute _ :: Statement → act .

execute ⟦ E1:Expression ":=" E2:Expression ⟧ =
    | evaluate E1 and then
    | coercively evaluate E2
    then
    | store the given item #2
    |       in the given cell #1 .

execute ⟦ "write" E:Expression ⟧ =
    coercively evaluate E then batch-send it .

execute ⟦ "read" E:Expression ⟧ =
    | batch-receive an integer and then evaluate E
    then
    | store the given integer #1
    |       in the given integer cell #2 .

execute ⟦ "if" E:Expression "then" S1:Statement
      "else" S2:Statement "endif" ⟧ =
    | coercively evaluate E
    then
    | | check it then execute S1
    | or
    | | check not it then execute S2 .

execute ⟦ "while" E:Expression "do" S:Statement
      "endwhile" ⟧ =
    unfolding
    | | coercively evaluate E
    then
    | | check it then execute S then unfold
    | or check not it .

execute ⟦ I:Identifier "(" E:Expression ")" ⟧ =
    | give the abstraction bound to id I and then
    | coercively evaluate E
    then
    | enact application the given abstraction #1
    |       to the given integer #2 .

execute ⟦ S1:Statement ";" S2:Statement ⟧ =
    execute S1 and then execute S2 .

execute "skip" = complete .
```

B.3.5 Expressions

```
• evaluate _ :: Expression → act .

evaluate "true" = give true .

evaluate "false" = give false .

evaluate i:Integer = give integer-value i .

evaluate I:Identifier = give the datum bound to id I .

evaluate ⟦ I:Identifier "[" E:Expression "]" ⟧ =
    | give the list bound to id I and then
    | | coercively evaluate E then give sum(it, 1)
    then
    | give component# (the given integer #2)
    |       items (the given list #1) .

evaluate
      ⟦ E1:Expression O:Operation E2:Expression ⟧ =
    | coercively evaluate E1 and then
    | coercively evaluate E2
    then give operation-result O .

evaluate ⟦ "not" E:Expression ⟧ =
    coercively evaluate E then
    give not it .
```

B.3.6 Operations

```
• operation-result _ ::
      Operation → dependent datum .

operation-result "+" =
    sum(the given integer #1,
        the given integer #2) .
```

```
operation-result "-" =
    difference(the given integer #1,
               the given integer #2) .

operation-result "<" =
    (the given integer #1) is less than
    (the given integer #2) .

operation-result "=" =
    (the given item | cell #1) is
    (the given item | cell #2) .

operation-result "and" =
    both(the given truth-value #1,
         the given truth-value #2) .
```

B.3.7 Integers

```
• integer-value _ :: Integer → integer .

integer-value n:natural = n .

integer-value ⟦ "-" n:natural ⟧ = negation n .
```

B.3.8 Identifiers

```
• 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] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Eightteenth Symposium on Principles of Programming Languages.* ACM Press, January 1991.

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

[4] William R. Bevier, Warren A. Hunt, J. Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5:411–428, 1989.

[5] Anders Bondorf. Automatic autoprojection of higher order recursive equations. In *Proc. ESOP'90, European Symposium on Programming.* Springer-Verlag (*LNCS* 432), 1990.

[6] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[7] 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.

[8] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.

[9] Mads Dam and Frank Jensen. Compiler generation from relational semantics. In *Proc. ESOP'86, European Symposium on Programming.* Springer-Verlag (*LNCS* 213), 1986.

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

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

[12] Joseph A. Goguen, James W. Thatcher, and Eric G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology, Volume IV.* Prentice-Hall, 1978.

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

[14] John Hannan. Making abstract machines less abstract. In *Proc. Conference on Functional Programming Languages and Computer Architecture.* Springer-Verlag *LNCS*, 1991.

[15] John Hannan. Staging transformations for abstract machines. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation.* Sigplan Notices, 1991.

[16] John Hannan and Dale Miller. From operational semantics to abtract machines. *Journal of Mathmatical Structures in Computer Science*, To appear, 1991.

[17] Warren A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5:429–460, 1989.

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

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

[20] Gilles Kahn. Natural semantics. In *Proc. STACS'87*. Springer-Verlag (*LNCS* 247), 1987.

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

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

[23] 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.

[24] 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.

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

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

[27] J. Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.

[28] 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.

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

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

[31] 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.

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

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

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

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

[36] Peter D. Mosses. Action semantics. Lecture Notes, Version 9 (a revised version is to be published by Cambridge University Press in the Series *Tracts in Theoretical Computer Science*), 1991.

[37] 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.

[38] 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.

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

[40] Flemming Nielson and Hanne Riis Nielson. Two-level functional languages. Draft book. To be published by Cambridge University Press, 1991.

[41] Hanne R. Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ-calculus. *Science of Computer Programming*, 10:139–176, 1988.

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

[43] Jens Palsberg. An automatically generated and provably correct compiler for a subset of Ada. In *Proc. ICCL'92, Fourth IEEE International Conference on Computer Languages*, 1992.

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

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

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

[47] 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.

[48] 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.

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

[50] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, September 1981.

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

[52] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, 1985.

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

[54] David A. Schmidt. Detecting stack-based environments in denotational semantics. *Science of Computer Programming*, 11:107–131, 1988.

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

[56] 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.

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

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

[59] 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.

[60] Mads Tofte. *Compiler Generators.* Springer-Verlag, 1990.

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

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

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

[64] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.

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