LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - The University of Edinburgh

Correctness Proofs of Compilers and Debuggers: an Overview of an Approach Based on Structural Operational Semantics

(Extended Abstract of Ph.D. Thesis)

by

Fabio Q B da Silva

LFCS Report Series
ECS-LFCS-92-233

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

September 1992

Correctness Proofs of Compilers and Debuggers: an Overview of an Approach Based on Structural Operational Semantics

(Extended Abstract of Ph.D. Thesis)

Fabio Q. B. da Silva
LFCS - Department of Computer Science
University of Edinburgh,
Scotland, U.K.,
e-mail: fabio@dcs.ed.ac.uk

August 27, 1992

Abstract

This paper is an overview of the main results presented in the author's Ph.D. thesis [dS92]. In this paper we study the use of semantics-based formal methods in the specification and proof of correctness of compilers and debuggers. We use a Structural Operational Semantics as the basis for the specification of compilers and propose a notion of correctness based on an observational equivalence relation [ST87]. We define program evaluation and a notion of evaluation step based on a Structural Operational Semantics and use these definitions as the basis for the specification of debuggers. Debugger correctness is then defined by an equivalence relation between a specification and an implementation of the debugger based on the bisimulation concept [Par81].

1 Introduction

This work examines two related issues in the implementation of programming languages: the specification and correctness proofs of compilers, and the specification and correctness proofs of debuggers. While this work focuses on the latter problem we use, for both problems, a *Structural Operational Semantics* of the programming language as the reference point for correctness. The correctness of compilers and debuggers are essential aspects of

any programming environment that uses such tools. In this sense, this work provides an underlying theory for the design of such environments.

The research on compiler design dates back to the early sixties. Since then, a vast literature has emerged describing techniques and tools for compiler writing (e.g. [Gri71, TS85, ASU86, PP92]). The problem of compiler correctness has also been widely studied in the past [MP67, Mor73, Mos79, TWW81, Pol81, CM86, Des86, Joy89, Sim90]. Therefore, we have a common understanding of what the problems of compiler specification and correctness involve: we must give a definition of a compiler and prove that the code it generates for each program executes consistently with the semantics of the programming language.

Since there exists some freedom in defining what is meant by "consistent execution of the compiled code", it is in this aspect that the works in the literature differ most. Our approach extends and improves previous work on this subject by presenting a notion of compiler correctness in which the execution of the code of a program must be *observationally equivalent* (in the algebraic specification sense) to the semantics of the program as defined by a Structural Operational Semantics.

In contrast with compilers, debuggers have received little attention from a theoretical point of view. However, debugging is an important phase in the development of programs accounting for a large percentage of the cost of this development [Jon77, Sho83]. Therefore, the problems of specifying debuggers and proving them correct also deserve a better understanding and a theoretical treatment. In this work we clarify the main problems involved in the specification and correctness proof of debuggers and propose a framework for their *specification*, *prototyping*, *implementation*, and *correctness proof*.

This framework is composed of three major aspects. First, a notion of *program evaluation* based on a Structural Operational Semantics of the programming language, from which we derive a notion of *evaluation step* of programs. This notion of program evaluation is the basis of a system for prototyping implementations of programming languages. Our approach to this problem differs from other approaches in the literature, e.g., the CENTAUR-TYPOL system [BCD+87, Des88] and the Animator Generator [Ber91], in that program evaluation is proved to be *sound* and *complete* with respect to an underlying mathematical meaning of the Structural Operational Semantics formalism.

Second, an *abstract characterisation of debuggers* that uses the notion of evaluation step discussed above. A concrete debugger according to this abstract characterisation is correct (by construction) with respect to the Structural Operational Semantics of the programming language from which the evaluation step was derived. Correctness here means that the evaluation of a program using the debugger produces the same results as the evaluation using the semantics of the programming language. We also design a *specification language* that can be used for defining and prototyping debuggers according to this abstract characterisation.

Other authors have also proposed the definition and prototyping of debuggers based

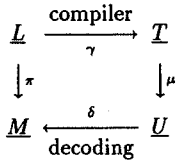

Figure 1: The Morris Diagram

on a semantics of the programming language [BS86, BMS87, Ber91, KHC91]. Mostly, they provide a pre-defined set of debugging capabilities that may be specialised to the particular programming language being used. In this work we do not fix the functionality of the debugger, but rather provide the means for the definition of this functionality using the specification language.

Finally, the third aspect of the framework we propose is a *notion of correctness* for an implementation of the debugger with respect to its specification. This notion of correctness is also an *observational equivalence* notion, in this case in the sense of a *bisimulation* between transition systems [Par81, Mil89]. As far as we are aware, the problem of correctness of debuggers has not been studied in the literature. Therefore, the solutions we propose to this problem are entirely novel to this work.

In this paper we give an overview of the framework for debugger design that is defined in author's Ph.D. Thesis [dS92]. Our objective is to introduce the main concepts of this approach by means of examples. The reader is referred to [dS92] for a formal treatment of such concepts.

2 The Design and Correctness of Compilers

In [MP67], McCarthy and Painter presented one of the earliest approaches to compiler correctness. This work consists of a proof of correctness of an algorithm for compiling arithmetic expressions into an abstract machine. In [BL69], Burstall and Landin introduced the use of algebraic methods in the compiler correctness problem. The use of an algebraic approach introduced structure on the objects involved in the correctness problem: programming language semantics, machine semantics, and the definition of the compiler.

The algebraic approach was further developed in [Mor73], where compiler correctness is characterised as the commutativity of the diagram in Figure 1 known as the Morris Diagram. In that diagram, the nodes are algebras and the arrows are homomorphisms. The π arrow denotes the semantics of the programming language, while the μ arrow denotes the semantics of the machine language.

In the Morris Diagram, the algebra $\underline{L}$ is the initial term algebra $T(\Sigma)$ for a signature Σ which defines the programming language. Therefore, γ and π are unique homomorphisms by initiality. The proof method is also based on initiality, for if μ and δ are also homomorphisms then the commutativity of the Morris Diagram follows by uniqueness. Therefore, a proof of compiler correctness consists of proving that the arrows of the diagram are homomorphisms.

One limitation of Morris' approach is that the correctness criterion requires the existence of a homomorphism δ from the algebra of the machine values into the algebra of the programming language values. To understand why this is a limitation let us consider a practical example. In functional languages, function expression evaluate to function values, or *closures*. For instance, the closure value of the expression fn x . x + 1 could be represented as $(\varepsilon_E, \mathrm{x}, \mathrm{x}+1)$, and the value of fn y . y + 1 could be represented as $(\varepsilon_E, \mathrm{y}, \mathrm{y}+1)$. It is conceivable that in a machine implementation these two expressions evaluate to the same machine value that "represents" all α-conversions of a closure like $(\varepsilon_E, \mathrm{x}, \mathrm{x}+1)$. In this case there is no homomorphism δ that makes the Morris Diagram commute.

The approach initiated by Morris inspired several investigations which set out to extend and improve the ideas presented in [Mor73]. We now discuss some of these investigations. In [TWW81], the ADJ group proposes the use of a homomorphism $\epsilon : \underline{M} \rightarrow \underline{U}$ (of encoding) to replace the homomorphism $\delta : \underline{U} \rightarrow \underline{M}$ in the Morris Diagram. A motivation for using ϵ is to overcome the limitations of the original diagram for cases like the function values discussed above.

However, the use of an encoding arrow in the correctness diagram is problematic in various ways. First, the commutativity of the diagram with $\epsilon : \underline{M} \rightarrow \underline{U}$ is not a sufficient criterion for correctness. For instance, in the case where $\underline{T}$ and $\underline{U}$ are one-point algebras and γ, μ, and ϵ are the unique homomorphism to these algebras, the diagram commutes trivially, as mentioned in [TWW81].

Another reason why ϵ does not give a sufficient correctness criterion is illustrated by a simple example. Suppose γ compiles every program $\underline{l}$ in $\underline{L}$ into the (fixed) code sequence $\underline{t}$ in $\underline{T}$. Therefore, since μ is a function, every program has the (fixed) meaning $\mu(\underline{l})$ in $\underline{U}$. Furthermore, for every $\underline{l}$ in $\underline{L}$ suppose that ϵ maps $\pi(\underline{l})$ into (the fixed) $\mu(\underline{l})$. The diagram then commutes trivially, although we intuitively would not regard this compiler as being correct.

The degenerate case of one-point algebras seems irrelevant in practice since we expect the machine language $\underline{T}$ never to be one-point. However, the second problem discussed above suggests that errors in the compiler arrow γ can be hidden by a suitable choice of the encoding arrow ϵ. Therefore, the use of the encoding arrow in the Morris Diagram is not adequate for compiler correctness.

Furthermore, the use of an encoding arrow suffers from a pragmatic problem. In practice, we use a compiler to translate a program into machine code; we then execute

the code on the machine and, if a result is produce by the execution, we expect to obtain its source level representation as the result of the program evaluation. In other words, we are interested in the results as they are represented in the algebra $\underline{M}$.

However, the existence of an encoding ϵ that makes the correctness diagram commute is not sufficient to guarantee we can (uniquely) convert from the machine representation of a result to its source language representation. In fact, a diagram with an encoding arrow only guarantees that there exists at least one result in $\underline{M}$ that corresponds to the result in $\underline{U}$ obtained from the execution of the program's code. We argue this is not sufficient from a pragmatic point of view.

A major limitation of the aforementioned algebraic approaches is that the algebraic semantics used to define the semantics of programming languages is functional. Therefore, these approaches do not directly deal with non-deterministic languages. In the approach initiated by Despeyroux [Des86] and followed by Simpson [Sim90], the nodes of the Morris Diagram become term algebras and the arrows become inductively defined relations between these algebras. Therefore, these approaches model non-deterministic languages naturally. The correctness diagrams used in [Des86, Sim90] use the encoding arrow ϵ. Therefore, both approaches suffer from the problems discussed above.

Some authors divide the compiler correctness problem into compiler *specification* correctness and compiler *implementation* correctness [Pol81, CM86]. The former refers to the correctness of a compiler specification with respect to the programming language semantics. This category includes all above cited works. Compiler implementation correctness refers to the correctness of a compiler implementation with respect to its specification.

Various aspects distinguish specification and implementation of compilers in this context. For instance, a specification is usually defined in terms of abstract syntax of the program while the implementation may involve lexical analysis, parsing, and so forth. Furthermore, a specification does not need to be executable, and even when it is executable, it is too inefficient to be used in practical applications. On the other hand, an implementation is necessarily executable and often robust for real applications.

This distinction was first addressed in [Pol81]. In [CM86] Chirica and Martin show how to apply the ideas of the Morris Diagram to prove correctness of a compiler implementation. Although we believe that compiler implementation correctness is an important problem, it is not addressed in our work.

Summarising the above discussion, we have seen that most approaches to the compiler correctness problem are based on the ideas proposed in [Mor73, TWW81]. However, the original Morris Diagram (see Figure 1) is too restrictive for some practical applications. Furthermore, the use of an encoding arrow ϵ replacing the decoding arrow δ cannot be considered a sufficient criterion for compiler correctness. A natural question at this point is whether there is a suitable generalisation of the Morris Diagram which is an (intuitively)

sufficient criterion and yet is general enough to address cases such as the closures.

Clearly, to require the encoding arrow ϵ to be injective gives a sufficient correctness criterion in the sense that it does not suffer from the problems addressed above. However, this restriction means that any two distinct program phrases with distinct semantics must have distinct target semantics. The example of the function values presented above shows that this restriction is too strong in some practical cases.

A less restrictive solution would be to use Hoare's idea of a representation relation [Hoa72] between the algebras $\underline{M}$ and $\underline{U}$. Another solution would be to compare the algebras $\underline{M}$ and $\underline{U}$ under observational equivalence [Rei81, ST87, NO88, Sch90]. The advantages of using observational equivalence over representation relation are discussed in [Sch87, page 255], where Schoett gives a proof that observational equivalence is more general than representation relation. Therefore, we propose to use observational equivalence as the criterion for compiler correctness.

Our approach to correctness is an improvement on previous approaches since it gives a more general, and yet intuitively sufficient, correctness criterion. Furthermore, it is based on a formal definition of equivalence which we can reason about at the meta-level. This level of reasoning is important since it is now possible to state and prove properties about the correctness criterion.

It is generally agreed that a major contribution of the ideas in [Mor73, TWW81, Pol81, CM86, Des86, Sim90] is that they present methodologies to structure the compiler and other semantic objects involved in the compiler correctness problem. However, this structure does not directly extend to the proofs of correctness which remains an ad hoc process. Various approaches have proposed ways of structuring the correctness proofs by using semi-automatic theorem provers [MW72, Coh78, Joy89, Sim90].

Another advantage of our approach to the compiler correctness problem is that we define a proof method based on correspondence relations [Sch87, Sch90]. This proof method is an improvement over ad hoc approaches because, besides being consistent with respect to observational equivalence, it introduces structure into the proofs of equivalence. This structure may suggest ways in which proofs can be semi-automated, contributing to the use of this framework in practical applications.

Summarising, our approach to compiler correctness affirms the ideas proposed in previous approaches and improves these ideas in various aspects. First, it gives a more general and yet (intuitively) sufficient criterion for correctness. Second, it provides a proof method which is consistent with respect to the correctness criterion. Finally, this proof method suggests a methodology to structure the proofs of correctness which complements previous advice on how to structure the other objects involved in the compiler correctness problem.

3 The Design and Correctness of Debuggers

In contrast with compilers, debuggers have received little attention from a theoretical point of view. Therefore, it is not commonly agreed what the problems involved in the design of debuggers are, nor what debugger correctness means. However, debugging is an important phase in the development of programs accounting for a large percentage of the cost of this development [Jon77, Sho83]. For instance, in the telecommunication industry this cost may account for over 50 percent of the total development cost of a program [Sev87]; similar figures have been reported from other areas.

Therefore, an important problem in software technology is to define methods and tools to reduce the time spent on the debugging phase [Lew82]. This reduction can be achieved either by producing programs that have fewer errors with respect to their specification (no errors in the ideal case), or by improving the quality of the tools and methods used in debugging, or by a combined solution.

The problem of developing programs that are correct with respect to their specification is the subject of a wide area of research on formal specification and formal program development. We do not treat this problem here; the interested reader is directed to [GM86] for an overview of various approaches to formal program development and to [BKL+91] for a survey on Algebraic Specification.

Although the research on formal specification experienced great advances in the last decade, it has not reached a state in which practical programs are developed entirely free of errors. Therefore debuggers are still necessary in the process of program development, and to improve the techniques and tools used in their design is an important problem to be addressed. We approach this problem by proposing the use of semantics-based formal methods in the design of debuggers. Our objective is to define a theory to address *formal specification*, *implementation*, and *correctness proofs* of such tools.

In the rest of this section we will study the process of *debugger design* and identify aspects in this process that can be improved by using semantics-based methods. Before we start looking into debugger design we should agree on what a debugger is and what distinguishes it from other programming tools. First, we are interested in *automated debuggers* rather than in manual debugging techniques like desk checking and memory dump analysis. As an initial proposal, we characterise debuggers as follows:

> A debugger is a tool that produces *information* about the *intermediate states* of the *evaluation* of programs under the *user's request.*

This characterisation emphasises that we are interested in *dynamic information* about a program rather than in a static analysis of its behaviour as in [Sev87]. We will refine and make this characterisation more precise throughout this work. Let us now analyse some questions that naturally arise from the above characterisation of debuggers:

- How are programs evaluated?
- What information about the evaluation may be requested?
- Is the process of requesting/obtaining information interactive?

The answers to these questions vary in the literature and define classes of debuggers. Programs may be interpreted, in which case the debugger is called an *interpreter-debugger*, as in [vdLW85, SYO87]. Alternatively, programs may be compiled into machine code that is then executed on the machine, in which case the debugger is known as a *compiler-debugger*. Finally, programs may be evaluated by a combination of interpretation and compilation, usually known as *mixed-execution* or *selective interpretation* [CH87].

The minimal information about the evaluation that is normally available is the value of the program data at intermediate states of the program evaluation. More sophisticated debuggers provide procedure and function trace-backs, and information about the control flow of the program. *Symbolic-debuggers* are debuggers in which this information is requested and presented at the level of the programming language structures. Finally, *interactive-debuggers* are those in which the process of requesting/obtaining information is interactive [Zel84, vdLW85, SYO87], in contrast with *post-mortem debuggers* in which the user cannot interact with the debugging process, and the information is delivered after program termination [Lau79].

Most proposals in the literature which set out to improve the quality of debugger focus on two main aspects: the *user interface* and the *information* that the user can access through the debugger. The recent advances in hardware technology and the wide availability of graphical work-stations has made possible the design of debuggers with sophisticated graphic interfaces [Bov87, Moh88]. The increasing computational power and storage capabilities of recent computers allows debuggers to store complete histories of the evaluation of the program, so that it is possible to access information about an arbitrary point in the history of the evaluation [Moh88].

The above mentioned works and related approaches in the literature (e.g. [vdLW85, SYO87]) have produced some improvements on the process of debugger design. However, this process still remains ad hoc and informal which often leads to debuggers that either have a *counter-intuitive behaviour*, or are *incorrect*, or both. Let us illustrate these problems so that we can understand how the use of semantics-based methods can assist in their solution.

Hereafter, we focus our interest on *interactive-debuggers*, for this is the most used kind of debugger and also because they provide the most interesting and challenging problems. In general, an interactive-debugger provides debugging commands that give the user the ability to step through the intermediate states in the evaluation of a program; to stop at desired break points; and to request information about the current, and possibly, previous states in the evaluation.

This characterisation of interactive-debuggers raises an important question: what is the meaning of "to step through the states in the evaluation of a program"? Answering this question involves the definition of a notion of an *evaluation step*, which is a central problem in the design of debuggers.

The importance of the definition of an evaluation step is brought to light in the example below in which we wish to emphasise how a *counter-intuitive* notion of evaluation step may lead to counter-intuitive behaviour of the debugger. In this case, the debugger is less effective in assisting the programmer to locate errors, and solutions to this problem are therefore necessary.

Suppose we have two versions of simple C [KR78] program in which the only difference is how the text of the program is written. The two versions of the program are shown below; we call the one on the left `p1.c` and the other `p2.c`:

```
main(){
  int i;
  i = 0;  while (1) {i++;};
}
```

```
main(){
  int i;
  i = 0;
  while (1) {i++;};
}
```

From our knowledge of the programming language, we develop an *intuition* about the evaluation steps of the above programs. According to this intuition, the first step in both evaluation is to assign `0` to the variable `i`. Then, the condition `(1)` of the `while` statement is tested. Since its value is different from `0`, the next step evaluates the statement `i++`, which increments the value of `i` by `1`. Since the value of `(1)` will always be different from `0`, the evaluation never leaves the `while` statement, and at each step the statement `i++` is evaluated, incrementing `i` by `1`.

The above description of the evaluation of `p1.c` and `p2.c` is what we call our *intuitive* notion of an evaluation step. When using a debugger, we expect to be able to observe the evaluation according to this intuitive notion. The following example shows a debugger, commonly used in practical applications, that does not behave as we expect.

Let us debug the programs `p1.c` and `p2.c` using DBX [Mic] to observe the values of the variable `i`. We first compile the programs generating debugging code, and then load DBX with this code. A debugging session for the program `p1.c` is shown below:

```
Reading symbolic information...
Read 34 symbols
(dbx) stop at 3
(2) stop at "p1.c":3
(dbx) run
Running: p1
stopped in main at line 3 in file "p1.c"
```

```
   3     i = 0;  while (1) {i++;};
(dbx) display i
i = 0
(dbx) step
^C
(dbx)
```

After the `step` command the execution enters an infinite loop and we cannot see the value of the variable `i` inside the `while` statement. A debugging session for the program `p2.c`, in which the first three debugging commands are as in the above session, is as follows:

```
Reading symbolic information...
Read 35 symbols
(dbx) stop at 3
(2) stop at "p2.c":3
(dbx) run
Running: p2
stopped in main at line 3 in file "p2.c"
   3     i = 0;
(dbx) display i
i = 0
(dbx) step
stopped in main at line 4 in file "p2.c"
   4     while (1) {i++;};
i = 0
(dbx) step
stopped in main at line 4 in file "p2.c"
   4     while (1) {i++;};
i = 2
(dbx)
```

Let us analyse two aspects of the above examples: the behaviour of the debugging command `step` in both debugging sessions; and the values output by DBX for the variable `i` during the evaluation of `p2.c`.

We start by analysing the `step` command. According to our intuition about the evaluation steps of `p1.c` and `p2.c`, we expect both programs to have identical evaluation steps, as we discussed on page 9. Therefore, we also expect the `step` command to have identical behaviours in both debugging sessions. Nevertheless, the behaviours of `step` do not agree with our intuition because DBX's notion of step is based on the lines of the program rather than on the sequence of primitive operations involved in the evaluation of the program.

We argue it is essential that a debugger behaves according to our intuition about how programs are evaluated. A debugger that behaves intuitively is easier to learn and may be more effective in assisting the programmer to locate errors. Therefore, an important problem in debugger design is to build debuggers with *intuitive behaviours.*

The particular problem we illustrated using the `step` command of DBX is an instance of a general problem. Let us characterise this problem to understand how semantic based methods may assist in its solution. On the one hand, as discussed above, we develop an intuitive notion of an evaluation step and consequently of how programs are evaluated. These intuitive notions form our *mental model* of the programming language [BOM81, Nor82] which is built mainly from the observations we make of program evaluations, and from the documentation we are given on the programming language, e.g, the definition of the programming language. On the other hand, a debugger also has a model of the programming language, which defines its notion of evaluation step and how programs are evaluated; we use the term *evaluation model* to refer to the debugger's model of the programming language. Problems therefore arise whenever our mental model and the debugger's evaluation model are different.

An obvious solution to this problem is to define debuggers whose evaluation model behaves close to our mental model. A first aspect in which semantics-based methods may help the design of debuggers is in the definition of an *intuitive evaluation model.* Some formalisms used to define semantic aspects of programming languages have an explicit operational meaning, e.g., definitional interpreters [Lan64] and Structural Operational Semantics [Plo81]. Other formalisms can naturally be given an operational interpretation, e.g., Natural Semantics [Kah88].

This operational meaning may be used to define evaluation models parametric on a formal semantics of the programming languages; when instantiated with a particular semantics such an abstract evaluation model yields an actual evaluation model that may be used to evaluate programs. For instance, this is the approach taken by Berry in his Animator Generator [Ber91].

Our starting point is that, using a suitable semantics formalism and a particular intuitive semantics of the programming language, we can define an evaluation model that has an intuitive notion of an evaluation step. Our main contention is that the use of such an evaluation model helps in the definition of debuggers that have *intuitive behaviour.*

Let us now analyse the values output by DBX for the variable `i` in the debugging session for program `p2.c` (page 10). In that debugging session, the value of `i` is shown as incrementing by two at each step. However, we intuitively expect it to be incrementing by just one. Since there is no formal specification of DBX, there are two possible interpretations for this behaviour. First, this is the behaviour intended by the debugger designer, in which case the behaviour of DBX is again counter-intuitive. Second, this behaviour is not intentional, in which case it is an *error* in DBX's implementation.

The second possibility is the most likely and is clearly the most harmful because if we use a debugger to locate errors in the program it is imperative that the debugger is *correct*. Another important aspect in debugger design is to produce *correct debuggers*.

The ability to build formal specifications is the first essential step towards correct debuggers. Another aspect in which semantics-based methods may assist in the debugger design is in the *formal specification* of the behaviours of the debuggers. A formal specification is an improvement on an informal one because it offers an unambiguous common reference for the users, the designers, and the implementors of the debugger, with the following advantages:

- A common reference between user and designer means that the behaviour of the debugger that the user learns is exactly what was specified by the designer.
- A common reference between the designer and the implementor means that the implementor may implement a debugger that has the behaviour specified by the designer in the formal specification. Moreover, because the specification is formal we can develop a notion of correctness between specification and implementation such that we can prove that the implemented debugger has the same behaviour as its specification.
- As a consequence of the two previous points, a common reference between the implementor and the user means that the latter uses a concrete debugger that behaves exactly like the specification.

Our second objective is to define a theory for the *specification of debuggers* based on a formal semantics of the programming language. In this semantics-based approach, an evaluation model of the programming language is used as the basis of the debugger specification. Since this evaluation model will be defined to behave as close as possible to our intuition about the programming language, we argue that our approach will assist in formalising debuggers with intuitive behaviours.

Our third and final objective is to define a notion of *equivalence* between the behaviours of a specification and an implementation of a debugger. The definition of an *evaluation model*, the methods and tools for *semantics-based specification* of debuggers, and the notion of *equivalence* between specification and implementation of debuggers form an integrated *theory of debugger design*. Our thesis is that the use of this theory helps in designing debuggers that more effectively assist the programmer because they are easier to learn, more intuitive to understand, and correct with respect to their formal specification. In the following sections we present an informal overview of this theory.

4 Choosing a Semantic Formalism

Our ability to construct formal specifications of debuggers depends on the existence of a formal notion of program evaluation and of evaluation step; we propose to use a formal semantics in the definition of programming languages and from such a semantics to derive these formal notions. The success of this proposal depends on the choice of a suitable semantic formalism, for on the one hand it is difficult to derive a notion of program evaluation from some formalisms (e.g., axiomatic semantics [Hoa69]). On the other hand, some formalisms are difficult to reason about because of the complexity of their underlying mathematics (e.g., denotational semantics [Sto89]). Using such a formalism could make it difficult to reason about debuggers.

However, some semantic formalisms have a simple underlying mathematics and a natural and explicit operational meaning that can be used in the definition of program evaluation. The generic name *operational semantics* is often used to refer to such a formalism. In fact, operational semantics is a class of formalisms that includes a diversity of styles of formal semantics: for instance, definitional interpreters (e.g., Landin's SECD machine [Lan64], or Milner's SMC machine [Mil76]), Plotkin's Structural Operational Semantics [Plo81], and Kahn's Natural Semantics [Kah88].

Structural Operational Semantics is a representative example of what we call the *transitional* style of operational semantics, in which the semantics of the programming language is defined by a transition system whose steps describe the evaluation of the programs. On the other hand, Natural Semantics is an example of what we call the *relational* style in which the semantics of a programming language is defined by a mathematical relation between programs and results.

The main difference between formalisms in the transitional and the relational styles that is relevant for our work at this stage is in the notion of an evaluation step. On the one hand, such a notion is an explicit component of formalisms in the transitional style. On the other hand, for formalisms in the relational style a notion of evaluation step must be defined as a component outside the formalism. This seems to imply that because our goal is a definition of an evaluation step for the debuggers, we should use a semantic formalism in the transitional style. This avoids the task of defining an explicit notion of evaluation step that is necessary if the chosen formalism is in the relational style of operational semantics.

Nevertheless, simple comparisons between concrete semantics written in Structural Operational Semantics and in a relational style, revealed that the latter provides semantics that are more concise and easier to reason about ([Ber91, pages 48–50], where what we call a Structural Operational Semantics is called a *transition semantics*). The use of *evaluation contexts* proposed in [WF91] helps in making Structural Operational Semantics more concise. However, this approach does not solve a limitation of the formalism: certain language constructors can only be defined by a Structural Operational

$$\frac{}{\mathbf{num}(n) \Rightarrow n}$$

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{\mathbf{plus}(e_1, e_2) \Rightarrow n_1 \ + \ n_2}$$

Figure 2: A Semantics of *Sum*

Semantics provided the language has a rich enough set of constructors. For instance, a `while` statement can only be defined provided the language has some kind of conditional statement.

We propose to use the relational style of operational semantics for the definition of semantic aspects of programming languages in this work. Therefore, we shall need to give an explicit definition of evaluation step to be used by the debuggers, as will be discussed later on.

Let us illustrate the semantic formalism that we shall use in the rest of this work, which we will call *Relational Semantics*. For this, suppose we have a simple language *Sum* of constant and sum expressions, defined by the following BNF rules:

$$\begin{array}{lcl} \mathit{exp} & ::= & \mathbf{num}(\mathit{nat}) \mid \mathbf{plus}(\mathit{exp}, \mathit{exp}) \\ \mathit{nat} & ::= & 0 \mid 1 \mid \cdots \end{array}$$

and a function $+ : \mathit{nat} \times \mathit{nat} \rightarrow \mathit{nat}$ that takes two *nat* numbers as arguments and returns a *nat* number that is the sum of its arguments. It is convenient to emphasise that **plus** is the language constructor whereas "+" denotes the mathematical sum operation. Moreover, **num** is a coercion operator that constructs an *exp* expression from a *nat* number. A Relational Semantics for the expressions in this language may be given by the set of inference rules in Figure 2.

The rules in Figure 2 inductively define a relation $\Rightarrow$ between *exp* and *nat*, and describe how to evaluate a *Sum* expression to a *nat* number: the first rule states that a constant expression $\mathbf{num}(n)$ evaluates to the *nat* number n; the second rule states that if the expression e_1 evaluates to a number n_1 and the expression e_2 evaluates to a number n_2 then the sum expression $\mathbf{plus}(e_1, e_2)$ evaluates to the result of $n_1 \ + \ n_2$. This informal interpretation is what we call an *operational interpretation* of the rules; such an interpretation is essential in the definition of an intuitive notion of evaluation step based on the Relational Semantics formalism, as will be discussed later on.

5 Equivalence of Definitions of Programming Languages

$$\frac{}{\mathbf{num}(n) \Rightarrow n}$$

$$\frac{e_1 \Rightarrow n_1 \qquad (n_1, e_2) \rightarrow n}{\mathbf{plus}(e_1, e_2) \Rightarrow n}$$

$$\frac{e_2 \Rightarrow n_2 \qquad (n_2, e_1) \rightarrow n}{\mathbf{plus}(e_1, e_2) \Rightarrow n}$$

$$\frac{e \Rightarrow n'}{(n, e) \rightarrow n + n'}$$

Figure 3: An Alternative Semantics for *Sum*

The same semantic aspect of a programming language may be defined by different semantic specifications. For instance, we could define the semantics of the language *Sum* by another set of rules as in Figure 3 (page 15). An important question is whether the set of rules in Figure 3 and Figure 2 define equivalent semantics for *Sum*, for if they do so we may use either of the two sets of rules to determine the result of a *Sum* expression. To answer this question we must first define what we mean by equivalent semantics. Using the operational interpretation of the two sets of rules, a suitable notion of equivalence for the above examples may be as follows:

> For all expressions e in *Sum*, e evaluates to a number n in the set of rules of Figure 2 if and only if e evaluates to n in the set of rules of Figure 3.

Given that the function "+" is commutative, a simple inspection of the two sets of rules tells us that they are equivalent in the above sense.

However, there are examples of pairs of semantics that we intuitively regard as being equivalent, but are not equivalent under the above informal notion. This suggests that we should look for a more general notion of equivalence. Let us show an example that illustrates this problem.

Consider a language *Pair* of pairs of numbers, defined by the following BNF rules:

$$pair \ ::= \ \mathbf{num}(nat) \mid \mathbf{cons}(pair, pair) \mid \mathbf{first}(pair) \mid \mathbf{second}(pair)$$

where *nat* was defined in the BNF rules of the language *Sum* (page 14). The sets of rules in Figure 4 (page 16) define two semantics for *Pair*. The set **Direct** defines the

Direct

$$\frac{}{\mathbf{num}(n) \hookrightarrow \mathbf{num}(n)}$$

$$\frac{p_1 \hookrightarrow p_1' \qquad p_2 \hookrightarrow p_2'}{\mathbf{cons}(p_1, p_2) \hookrightarrow \mathbf{cons}(p_1', p_2')}$$

$$\frac{p \hookrightarrow \mathbf{cons}(p_1, p_2)}{\mathbf{first}(p) \hookrightarrow p_1}$$

$$\frac{p \hookrightarrow \mathbf{cons}(p_1, p_2)}{\mathbf{second}(p) \hookrightarrow p_2}$$

Reverse

$$\frac{}{\mathbf{num}(n) \hookrightarrow \mathbf{num}(n)}$$

$$\frac{p_1 \hookrightarrow p_1' \qquad p_2 \hookrightarrow p_2'}{\mathbf{cons}(p_1, p_2) \hookrightarrow \mathbf{cons}(p_2', p_1')}$$

$$\frac{p \hookrightarrow \mathbf{cons}(p_1, p_2)}{\mathbf{first}(p) \hookrightarrow p_2}$$

$$\frac{p \hookrightarrow \mathbf{cons}(p_1, p_2)}{\mathbf{second}(p) \hookrightarrow p_1}$$

Figure 4: Two Semantics for *Pair*

intuitive semantics in which a *pair* expression is evaluated to another *pair* expression by removing all **first** and **second** constructors. The set **Reverse** gives a semantics in which a *pair* expression is also evaluated to another *pair* expression without **first** and **second** constructors, but the *pair* expressions of the form $\mathbf{cons}(pair, pair)$ are constructed in reverse order.

According to the notion of equivalence discussed above, **Direct** and **Reverse** do not define equivalent semantics for the language *Pair*. For instance, the *pair* expression $\mathbf{cons}(\mathbf{num}(1), \mathbf{num}(2))$ evaluates to $\mathbf{cons}(\mathbf{num}(1), \mathbf{num}(2))$ in **Direct** and evaluates to $\mathbf{cons}(\mathbf{num}(2), \mathbf{num}(1))$ in **Reverse**. However, $\mathbf{first}(\mathbf{cons}(\mathbf{num}(1), \mathbf{num}(2)))$ evaluates to $\mathbf{num}(1)$ in both sets of rules. Thus, if we are only interested in results of the form $\mathbf{num}(n)$ then the difference between **Direct** and **Reverse** becomes irrelevant, in which case we would like to consider the two sets of rules as equivalent semantics of *Pair*. Therefore, we need a more flexible notion of equivalence in which only a subset of the *pair* expressions are required to evaluate to the same result under the two sets of rules, namely those expressions that evaluate to some result of the form $\mathbf{num}(n)$.

Instances of the above problem arises naturally in the semantics of practical programming languages. For instance, in languages with abstract data types, we may not be interested in the details of the representation of abstract data type values. In functional languages, we are often not interested in the details of the representation of function values. Therefore, semantics that choose different representation for such values can still be considered equivalent if we use a suitable notion of equivalence.

The idea of regarding only a subset of the results of expressions as being relevant, or observable, is known in the area of algebraic specification as observational or behavioural equivalence between algebras. However, a similar notion already appears implicitly in

the simulation method introduced by Milner [Mil71]. This idea applied to algebras first appears as the concept of the "semantics" of an algebra in [GGM76].

In our approach we propose a notion of equivalence that has the intuitive meaning discussed above; this particular notion of equivalence is based on *observational equivalence* as defined in [ST87]. In Chapter 3 of [dS92], we extend the definition of *strong correspondence relations* given in [Sch87, Sch90] to the formalism of Relational Semantics to obtain a practical proof method for proofs of equivalence between semantics.

6 Compiler Correctness

In order to motivate the following discussion about the compiler correctness problem let us consider a simple stack machine *M-Sum* which has a stack on which we can push *nat* numbers, an accumulator that is used for arithmetic operations, and three instructions that manipulate the contents of the stack and the accumulator: **push** that pushes the value of the accumulator on the top of the stack; **add** that adds the top element of the stack to the accumulator, leaving the result in the accumulator; and **load** that loads a *nat* number into the accumulator. The syntax of the machine language is given by the following BNF rules:

$$\begin{array}{lll} \mathit{stack} & ::= & \mathbf{empty} \mid \mathit{nat} \cdot \mathit{stack} \\ \mathit{inst} & ::= & \mathbf{push} \mid \mathbf{add} \mid \mathbf{load}(\mathit{nat}) \end{array}$$

A program in this machine is a sequence of instruction, for instance,

$$\langle \mathbf{load}(1), \mathbf{push}, \mathbf{load}(2), \mathbf{add} \rangle$$

The semantics of this machine may also be defined using a Relational Semantics. In this definition, a machine state is a pair (S, n) where S is a stack and n is the accumulator. The rules in Figure 5 define how machine instructions operate on a machine state to produce the new values of the stack and accumulator. In that rules, the concatenation of two sequences of instructions i_1 and i_2 is denoted by $i_1 @ i_2$.

For instance the *M-Sum* program $\langle \mathbf{load}(1), \mathbf{push}, \mathbf{load}(2), \mathbf{add} \rangle$ evaluating on a state $(\mathbf{empty}, 0)$ produces the state $(\mathbf{empty}, 3)$. This machine may be used as the target machine of a compiler for the language *Sum*. The objective is to translate expressions into machine programs that evaluate to the same result. For instance, the expression $\mathbf{plus}(1, 2)$ could be translated into the program $\langle \mathbf{load}(1), \mathbf{push}, \mathbf{load}(2), \mathbf{add} \rangle$.

A compiler for *Sum* using *M-Sum* as the target machine could be defined by the following set of rules that describes how *Sum* expressions are translated into sequence of *M_Sum* instructions:

$$\frac{}{\mathbf{num}(n) \rightsquigarrow \langle \mathbf{load}(n) \rangle}$$

$$\frac{}{(S,n) \vdash \langle \mathbf{push} \rangle \rightarrow (n \cdot S, n)}$$

$$\frac{}{(n_1 \cdot S, n_2) \vdash \langle \mathbf{add} \rangle \rightarrow (S, n_1 + n_2)}$$

$$\frac{}{(S,n) \vdash \langle \mathbf{load}(n') \rangle \rightarrow (S, n')}$$

$$\frac{(S,n) \vdash i_1 \rightarrow (S', n') \qquad (S', n') \vdash i_2 \rightarrow (S'', n'')}{(S,n) \vdash i_1 @ i_2 \rightarrow (S'', n'')}$$

Figure 5: A Semantics for *M-Sum*

$$\frac{e_1 \rightsquigarrow c_1 \qquad e_2 \rightsquigarrow c_2}{\mathbf{plus}(e_1, e_2) \rightsquigarrow c_1 @ \langle \mathbf{push} \rangle @ c_2 @ \langle \mathbf{add} \rangle}$$

A definition of the semantics of *Sum* using this compiler may be given by the following rule, which defines how an expression e is evaluated into a *nat* number:

$$\frac{e \rightsquigarrow c \qquad (\mathbf{empty}, 0) \vdash c \rightarrow (\mathbf{empty}, n)}{e \Rightarrow n}$$

The above rule has the following operational interpretation: if a *Sum* expression e compiles to a *M-Sum* machine program c and this program, runing on the state $(\mathbf{empty}, 0)$, produces a state $(\mathbf{empty}, n)$ then the result of the evaluation of e is n. We use the term *Evaluation by Compilation* to refer to a semantics of a programming language given via its translation into some target machine code. In our approach, the compiler defined by the relation $\rightsquigarrow$ is considered *correct* if the above definition of the relation $\Rightarrow$ is equivalent to the definition of $\Rightarrow$ of Figure 2. We use the relation of observational equivalence discussed in Section 5 as the notion of compiler correctness.

Compiler correctness has been widely studied in the past (e.g. in [MP67, Mor73, Mos79, TWW81, Pol81, CM86, Des86, Sim90]). Our approach strengthens and improves these previous approaches in two main aspects. First, it gives a more general, yet intuitively sufficient, criterion for correctness. Second, we define a proof method that can be used in practical proofs of correctness.

7 Program Evaluation and a Notion of Evaluation Step

We discussed above that a Relational Semantics of a programming language may be used to evaluate programs in the language, but we have not yet seen how this may be done. Let $\mathbf{num}(1) + \mathbf{num}(2)$ be a *Sum* expression that we want to evaluate using the semantics of Figure 2. By viewing the rules as an inductive definition of the relation $\Rightarrow$ between *exp* expressions and *nat* numbers, we can build a proof tree [DF87] for the formula $\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow n$, where n is a meta-variable. The process of constructing such a tree finds an instantiation for n that is the result of the evaluation of $\mathbf{num}(1) + \mathbf{num}(2)$.

We are not interested in the details of how to build proof trees. However, let us show the complete proof tree for $\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow n$ as an illustration:

$$\frac{\dfrac{}{\mathbf{num}(1) \Rightarrow 1} \qquad \dfrac{}{\mathbf{num}(2) \Rightarrow 2}}{\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2)) \Rightarrow 3}$$

where $1 + 2 = 3$. The root of this tree is an instance of the rule that defines **plus** in Figure 2, and the leaves are instances of the rule for **num** in the same figure.

Proof tree construction is a possible method for evaluating programs using a Relational Semantics of the programming language. An implementation of an algorithm for constructing proof trees that is parametric on a set of rules provides an abstract evaluation model that may be used to generate concrete evaluation models, or interpreters, for programming languages based on their semantics. This idea was first used in the CENTAUR system [BCD+87], and has also been used in other more recent works [Chi89, Ber91].

In principle, we could use such an evaluation model as the evaluation model of the programming language to be used by debuggers, as discussed in Section 3. Then, the only remaining problem would be to define a notion of evaluation step based on this evaluation model. In [Ber91], Berry defines such a notion of evaluation step as a function between partial proof trees (proof trees in which some sub-trees are not constructed); this function characterises a depth-first left-to-right traversal of a proof tree.

However, Berry's definition of evaluation step involves various complex definitions that make it difficult to understand and reason about, so cluttering the intuitive understanding of the notion of step. This complexity is necessary in that case because of the requirements of Berry's "theory of program animation". Debuggers in our framework do not have such requirements; thus we should seek a simpler notion of evaluation step that is intuitive and easier to reason about.

We propose an evaluation model for programming languages inspired by the SOS semantics of [And91]. This evaluation model is parametric on a Relational Semantics and is defined by a transition system whose states, called *evaluation states*, contain the current state of the evaluation of the program (i.e., the result of evaluated sub-programs) and the sub-programs or *goals* that still need to be evaluated. A transition step of this system uses the rules of the semantics to decompose a current goal into sub-goals to update the current

state and advance the evaluation. For instance, suppose that $\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2))$ is the expression to be evaluated; an initial state of the transition system is as follows:

$$[\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2)) \Rightarrow n]$$

In this evaluation state, $\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2)) \Rightarrow n$ is the goal. A transition step from this state advances the evaluation as follows:

$$[\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2)) \Rightarrow n] \stackrel{CS}{\Rightarrow} [n = n_1 + n_2 : \mathbf{num}(1) \Rightarrow n_1, \mathbf{num}(2) \Rightarrow n_2]$$

where $\stackrel{CS}{\Rightarrow}$ denotes a transition of the system, and $n = n_1 + n_2$ indicates that the meta-variable n is substituted by the term $n_1 + n_2$ in the current state. A substitution of terms for meta-variables is an explicit part of every state of the transition system. We omit the irrelevant parts of the substitutions from the states to simplify the presentation of the examples.

In the above transition, the resulting state indicates that the next goal is to evaluate $\mathbf{num}(1)$ producing a result n_1, then to evaluate $\mathbf{num}(2)$ producing a result n_2, and finally to compute the sum $n_1 + n_2$. This transition is defined using the rule for **plus** expression of Figure 2 to expand the goal $\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2)) \Rightarrow n$ into the sub-goals $\mathbf{num}(1) \Rightarrow n_1$ and $\mathbf{num}(2) \Rightarrow n_2$, each of which corresponds to a premiss of the rule. The next step in the evaluation is as follows:

$$[n = n_1 + n_2 : \mathbf{num}(1) \Rightarrow n_1, \mathbf{num}(2) \Rightarrow n_2] \stackrel{CS}{\Rightarrow}$$
$$[n = 1 + n_2, n_1 = 1 : \mathbf{num}(2) \Rightarrow n_2]$$

which is defined using the rule for the definition of $\mathbf{num}(n)$ expressions of Figure 2. After this transition the resulting state indicates that $\mathbf{num}(1)$ has been evaluated to 1 and $\mathbf{num}(2) \Rightarrow n_2$ is the current goal. Finally, we can apply a transition step to obtain the final state of the evaluation and the result of the sum:

$$[n = \mathbf{num}(1) + \mathbf{num}(n_2), n_1 = 1 : \mathbf{num}(2) \Rightarrow n_2] \stackrel{CS}{\Rightarrow}$$
$$[n = 1 + 2, n_1 = 1, n_2 = 2 : \varepsilon]$$

where ε indicates that there are no more sub-goals and the evaluation terminated successfully. The substitution in the final state gives the result of the evaluation, in this case $n = 1 + 2 = 3$.

In Chapter 5 of [dS92] we formalise the transition system discussed above, which we call the *Computational Semantics* of the Relational Semantics formalism. This transition system is used as the evaluation model for programming languages; the notion of evaluation step used by the debugger is given by the transition relation of this transition system. In [dS92], we give some evidence that this notion of evaluation step is indeed intuitive. Therefore, this transition system and the formalisation of Relational Semantics form the basic components of a theory of debugger design.

This transition system defines an evaluation model for a wide class of programming languages, including non-deterministic languages. Moreover, it defines a deterministic evaluation model for deterministic languages that may be implemented to yield an evaluation model generator for such languages. This provides a prototyping facility for programming languages similar to that provided by the CENTAUR system [BCD+87] and the Animator Generator [Ber91]. The advantages of our transition system over other methods in the literature is that it has an explicit, intuitive notion of an evaluation step.

8 Formal Specification of Debuggers

The next problem we shall address is how to construct formal specifications of debuggers using the evaluation model discussed above. Let us motivate the following discussion by considering the problems involved in specifying a debugger for the language *Sum*. The first step in this specification is the definition of the *debugging language*, i.e., the language in which we write debugging commands. The following BNF rules define a simple debugging language, called *Pico*:

deb ::= **step** | **trace on** | **trace off**

Recall that using the evaluation model described above we can evaluate *Sum* expressions step by step. Therefore, a possible interpretation for the above debugging language is as follows: **step** causes the evaluation of the current expression to advance by one transition step of the evaluation model; **trace on** switches on tracing mode; **trace off** switches off tracing mode. When tracing mode is on a **step** command outputs the result of the last sub-expression evaluated and the sub-expression that is about to be evaluated. Our goal is to discuss how this informal interpretation may be formally defined, leading to a specification of the debugger.

There are at least two ways in which to formalise the definition of a particular debugger: in an ad hoc fashion, or by first defining a formal notion of an abstract debugger, and then giving a definition of the concrete debugger that conforms with this abstract notion. The advantages of the second method are that we have an abstract notion of debuggers that can be formally reasoned about, as well as a formal definition of the concrete debugger that is also amenable to formal reasoning. For instance, the definition of equivalence between the behaviour of two arbitrary debuggers, discussed later on, depends on such an abstraction.

To achieve an abstract notion of debuggers we need a more elaborated characterisation of debuggers than that given on page 7:

> A debugger is a tool that receives *debugging commands* from the user, applies those commands to its current *debugging state*, and outputs the information

about the evaluation of the program requested by the command as its *result*. In this process the debugger may advance the current state of the evaluation, which is a component of its current debugging state.

In Chapter 6 of [dS92] we formalise the above characterisation leading to an abstract formal notion of debuggers. We also discuss various aspects of the design of concrete debuggers that conform to this notion. In this abstract notion a debugger is a monogenic labelled transition system whose states are the *debugging states* and the labels of the transition relation are the *debugging commands* and their *results*. An evaluation state of a program, according to the evaluation model of the programming language, is a component of each debugging state.

This view of a debugger as a labelled transition system is implicit in the event-action model of debugging proposed by some authors in the literature [BH83, GB85, LL89]. Our work improves on those ideas by the use of formal semantics in the definition of the programming language, an evaluation model of the programming language in the evaluation of programs, and by developing an abstract definition of debuggers that is amenable to formal reasoning at the level of an arbitrary debugger.

A possible definition of the states of the debugger *Pico* is a pair of the form (h, tr) where: h is a sequence of evaluation states the evaluation model of the *Sum* language and tr is a boolean variable used as the tracing mode flag. If we want to debug the *Sum* expression $\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2))$, we load the debugger with this expression, which causes the debugger to build an initial debugging state of the form:

$$(\langle[\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2)) \Rightarrow n]\rangle, \mathbf{false})$$

and start the debugging session. A debugging session is an interactive process in which we input debugging commands and the debugger outputs the results; in this process we have no access to the debugging states. If we issue a command **trace on**, the resulting debugging state is:

$$(\langle[\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2)) \Rightarrow n]\rangle, \mathbf{true})$$

If we issue a **step** command on this state, the resulting debugging state becomes:

$$(\langle[\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2)) \Rightarrow n], [\mathbf{num}(1) \Rightarrow n_1, \mathbf{num}(2) \Rightarrow n_2]\rangle, \mathbf{true})$$

with output: **no previous result.**
current expression: $\mathbf{num}(1)$

The evaluation state $[\mathbf{num}(1) \Rightarrow n_1, \mathbf{num}(2) \Rightarrow n_2]$ is obtained from the previous state $[\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2)) \Rightarrow n]$ using $\stackrel{cs}{\Rightarrow}$. Here we omitted the substitution of $n_1 + n_2$ for n to simplify the presentation. If we issue another **step** command on the above debugging state, the state changes as follows:

$$(\langle[\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2)) \Rightarrow n], [\mathbf{num}(1) \Rightarrow n_1, \mathbf{num}(2) \Rightarrow n_2], [\mathbf{num}(2) \Rightarrow n_2]\rangle, \mathbf{true})$$

with output: **previous result:** $n_1 = 1$
current expression: num(2)

The debugging session continues until the expression is fully evaluated or we abort the evaluation.

The debugger *Pico* only changes the *st* part of a debugging state (st, tr) by using the evaluation model $\stackrel{CS}{\Rightarrow}$. For this reason we say that debuggers in our framework are semantic-driven or *based on formal semantics.* This characteristic is formalised in [dS92] as a requirement every debugger must fulfill. Therefore, there is a repertoire of debugging commands in existing debuggers that are disallowed in our framework, e.g., changing the value of programming language variables. If such commands are allowed, a debugger could generate evaluation states that are not reachable from the initial evaluation state of a program using $\stackrel{CS}{\Rightarrow}$. In this case, we could not regard such a debugger as being *based on formal semantics.*

Once we have developed an abstract characterisation of debuggers, the next problem is to study how we may formally specify concrete debuggers that conform to this characterisation. This problem is studied in two stages: we study some aspects of the specification of debuggers and then define a notation to assist in the specification of concrete debuggers. The design aspects studied are generic and the proposed solutions to the problems may be applied to any concrete debugger that uses the evaluation model described above in the evaluation of programs.

The specification notation, called DSL, is defined in [dS92] with the objective of making the specification of concrete debuggers simpler and less ad hoc. This is achieved because DSL provides a high-level abstraction of the objects of the evaluation model, and also a powerful set of constructs to express debugging commands. DSL is a specification notation in the sense that definitions written in the language are abstract and concise.

Our major goal in the definition of DSL is to achieve an expressive language in which a useful set of debugging commands, found in most practical debuggers can be specified. In [dS92], we present some examples that show that DSL is indeed expressive. Nevertheless, the theory of debugger design does not depend on this particular design language; other languages may be defined and used with the theory to suit the needs of particular debuggers.

The notion of an abstract debugger and the DSL notation form a basic framework for the specification of concrete debuggers. This framework is based on the notion of program evaluation and evaluation step discussed above; thus the framework is formal with respect to the semantics of the programming language. Our goal is to show that this framework helps to define debuggers with intuitive behaviours; definitions that can be used as a documentation for the user and guides for implementors.

9 Implementation and Correctness of Debuggers

In the debugger *Pico*, programs are interpreted using the evaluation model $\stackrel{CS}{\Rightarrow}$. This definition of *Pico* can be viewed as a prototype of the debugger which is simple to specify and understand. Prototypes are useful for testing the functionality of the specified debuggers but they seldom have the performance required in practical applications. Therefore, there is the need for implementations of the debuggers that meet such practical requirements in performance. In our approach an implementation is another specification of the debugger given at a lower level of abstraction which is often more efficient than the original specification.

As discussed in Section 3, the main purpose of formal specifications of debuggers is to serve as a common unambiguous reference for the users, the designers, and the implementors. It is therefore necessary that the implementation of a debugger behave as defined by its specification. The main problem discussed below is how to establish the conditions for an implementation to be correct with respect to a specification of a debugger.

To illustrate the problems involved in implementing debuggers, and their correctness, let us first define another debugger, called *M_Pico*, that has the same debugging language as the debugger *Pico*. The debugger *M_Pico* uses the compilation of *Sum* expressions into *M_Sum* code to evaluate expressions. Therefore, the debugging commands of *M_Pico* work on the machine states instead of on the states of the evaluation model for *Sum*. For this reason, *M_Pico* is what we call a *compiler-debugger*.

Let us consider a modified definition of the machine states in which we add the code that is executing as a component; we write this new state as:

$$(S, \mathrm{Acc}) \vdash c$$

where (S, Acc) is the state of the machine as defined on page 17, and c is a *M_Sum* program. An *M_Pico* debugging state is a pair (m_st, tr) where m_st is a machine state as above and tr is a tracing mode flag. In *M_Pico* **step** advances the evaluation by one machine step, which corresponds to the evaluation of one machine instruction. The command **trace on** and **trace off** work as in *Pico*, with the difference that here the result of the last sub-expression is the value loaded into the accumulator by the previous instruction and the current sub-expression is the instruction that is about to be evaluated.

If we want to debug the *Sum* expression $\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2))$ we load *M_Pico* with the expression; the debugger calls the compiler that translates the expression into its corresponding *M_Sum* code. Finally, the debugger creates the initial debugging state:

$$((\mathbf{empty}, 0) \vdash \langle \mathbf{load}(1), \mathbf{push}, \mathbf{load}(2), \mathbf{add} \rangle, \mathbf{false})$$

and starts the debugging session. We can switch on tracing mode by issuing a **trace on**

command, such that the next debugging state is:

$$((\mathbf{empty}, 0) \vdash \langle \mathbf{load}(1), \mathbf{push}, \mathbf{load}(2), \mathbf{add} \rangle, \mathbf{true})$$

If we issue a **step** command on this state the resulting debugging state is:

$$((\mathbf{empty}, 1) \vdash \langle \mathbf{push}, \mathbf{load}(2), \mathbf{add} \rangle, \mathbf{true})$$

with output: **previous result: Acc = 1**
current instruction: push

Another **step** command produces the debugging state and the output below:

$$\langle (1 \cdot \mathbf{empty}, 1) \vdash \langle \mathbf{load}(2), \mathbf{add} \rangle, \mathbf{true} \rangle$$

with output: **no previous result.**
current instruction: load(2)

Here, since **push** does not load a value into the accumulator, we do not have a previous result. This process continues until the evaluation of the code finishes or we abort the debugging session.

Although *M_Pico* can be used to debug *Sum* expressions we could hardly regard it as a correct implementation of *Pico* in the sense of *having the same behaviour*. The two major reasons for the different behaviours of the two debuggers are:

- There are more steps in the evaluation of the expression according to *M_Pico* than there are using *Pico*; we need to issue four **step** commands in *M_Pico* to entirely evaluate the above expression, whereas it is only necessary to issue three **step** commands in *Pico*.
- *M_Pico* shows machine registers and instructions as the information about the evaluation whereas *Pico* shows source language expressions and results.

The possibility that differences arise between an implementation of a debugger and its specification should be ruled out by a notion of correctness between implementation and specification. If we consider that an implementation of a debugger is also defined as a labelled transition system then we may informally characterise a notion of correctness as an equivalence between debuggers, as follows:

> Two debuggers are equivalent if there exists a one-to-one correspondence between the states of the debuggers such that whenever the two debuggers are at corresponding states each debugging command produces equal results and the debuggers move to corresponding states.

Requiring a one-to-one correspondence between the debugging states rules out the first difference, and requiring that outputs from the same command at corresponding states must be equal makes sure that the implementation will not show machine registers and instructions if the specification does not do so. This notion is sufficient as far as the two aspects discussed above are concerned, but requiring equality between results may be too strong.

Suppose that we define two debuggers for the language *Pair* of Figure 4 with the same debugging language as *Pico*: the debugger called *D* evaluates *pair* expressions using the *Direct* semantics; the debugger *R* evaluates *pair* expressions using *Reverse*. It is not difficult to check that the states of the evaluation in both semantics are in a one-to-one correspondence. However, suppose that at some state in the evaluation of a *pair* expression the result of the last sub-expression evaluated is a *pair* of the form **cons**(*pair*, *pair*) then, since *Reverse* implements such a pair in reversed order the two debuggers will not be considered equivalent according to the above definition.

As in the problem of equivalence between definitions of programming languages, if we are not interested in the representation of *pair* results of the form **cons**(*pair*, *pair*) then we would like to regard the two debuggers as being equivalent. Therefore, we need a notion of equivalence between debuggers that compares the result of debugging commands up to observational equivalence.

We propose a notion of equivalence between debuggers, inspired by the bisimularity concept of [Par81] and strong congruence of [Mil89]. We extend such ideas so that labels of transitions are only compared up to observational equivalence. Therefore, equivalence between programming language definitions – in particular, compiler correctness – and equivalence between debuggers are strongly related. This relationship is formally stated in various results in [dS92].

These definitions constitute the last components of the theory of design of debuggers. Therefore, in this theory, debuggers may be formally specified using the DSL language and implementations of debuggers may be formally defined and proved correct with respect to their specifications. The examples in [dS92] have the objective of demonstrating that, using the theory:

- It is practical to formally define debuggers that have intuitive behaviour.
- It is practical to prove the correctness of debugger implementations.

10 Compiler-Debuggers

Another important problem in the design of debuggers is the study of *Compiler-debuggers* and the problems related with their design. We use the name Compiler-debugger for debuggers in which programs are compiled into machine code and debugging is performed

on the execution of the code on the machine. In such a debugger, the debugging commands must be defined on the machine states instead of on a state of the evaluation model of the language; *M_Pico* is an example of a Compiler-debugger.

In Chapter 7 of [dS92] we characterise Compiler-debuggers and study various aspects of their design. In this study we clarify the main aspects in which a Compiler-debugger differs from a specification based on the evaluation model of the programming language, and investigate mechanisms that may be used to resolve these differences. The objective is to define Compiler-debuggers that are correct with respect to a specification of the debugger. The study of the design of Compiler-debuggers clarifies two aspects that were already discussed above:

- Since we expect the evaluation of the machine code of a program to involve more steps than its interpretation under the evaluation model, how can we establish a one-to-one correspondence between those two notions of evaluation step?
- How do we recover the information about the evaluation that is needed by the debugging commands from a machine state and output the results in a source language form?

We treat these problems at the level of an abstract Compiler-debugger so that the proposed solutions may be used with any such a debugger. We also consider other aspects of Compiler-debuggers, e.g., optimiser-debuggers [Hen82, ZJ91], and show that these aspects may have a formalisation in the theory defined in our approach.

11 Concluding Remarks

This paper is an overview of the main results presented in [dS92]. We did not give formal definitions, but rather illustrated these results by examples. In [dS92], we set out to examine the problems in the specification and the correctness proofs of compilers, and in the specification and the correctness proofs of debuggers. We proposed a framework that addresses specification, prototyping, implementation – in the sense of low level specification, and correctness proofs of such tools using a particular characterisation of Structural Operational Semantics as the underlying formalism.

The main results of our approach are: a definition of a variant of Structural Operational Semantics, called Relational Semantics, which is the underlying formalism of our approach; the definition of a notion of observational equivalence between Relational Semantics Specifications; a formulation of the problem of compiler correctness using observational equivalence; an evaluation model for programming languages and a definition of an evaluation step; an abstract definition of Interpreter-debuggers; a specification notation for the formal specification of debuggers, called DSL; a notion of equivalence between

debuggers using bisimulation; a study on Compiler-debuggers and the problems involved in their definition.

The main contributions of these results are related to debugger correctness. We demonstrated that it is possible to give formal specifications of debuggers based on a formal semantics of the programming language. In practice, a given debugger may have more than one specification, tailored to different purposes. Related to this topic we examined the problem of Compiler-debuggers as an alternative specification of debuggers. We demonstrated that it is possible to prove the equivalence of specifications of a debugger, and, in particular, that it is possible to prove the correctness of Compiler-debuggers.

As far as we are aware of, the problems of specifying Compiler-debuggers, and of debugger correctness, have not been addressed before in the literature. Moreover, no other related work proposes a theory that addresses compiler and debugger correctness uniformly. These are the main, novel results of our work. We hope that these results will assist in the design of compilers and debuggers in practice, and also provide the basis of further research on the specification and correctness of other programming tools.

In [dS92], we discuss various directions in which our approach can be improved and extended. A brief summary of these directions includes: extensions on the semantic formalism to include Relational Rules with negative premisses; specification of debuggers for concurrent and non-deterministic languages; various aspects of our framework that can be given a machine implementation.

Acknowledgements

The author would like to thank his supervisors in Edinburgh, Kevin Mitchell and Don Sannella, for helpful suggestions on the research that led to the results presented in this paper. Thanks also to Ed Kazmierczak, Kees Goossens, Matthew Morley, and Dave Berry for proof-reading previous versions of this text. The author is supported by a Brazilian government scholarship, CNPq process number 20.0459/88.0/CC.

References

[And91] James H. Andrews. *Logic Programming: Operational Semantics and Proof Theory.* PhD thesis, LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, July 1991.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Publishing Company, 1986.

[BCD+87] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. Technical Report 777, INRIA, Rocquencourt, France, December 1987.

[Ber91] D. Berry. *Generating Program Animators from Programming Language Semantics.* PhD thesis, LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, June 1991.

[BH83] B. Bruegge and P. Hibbard. Generalized path expressions: a high level debugging mechanism. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 34–44, August 1983. ACM Software Engineering Notes 8(4); ACM SIGPLAN Notices 18(8).

[BKL+91] M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella, editors. *Algebraic System Specification and Development*, volume 501 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[BL69] R. M. Burstall and P. J. Landin. Programs and their proofs: an algebraic approach. *Machine Intelligence*, 4:17–43, 1969.

[BMS87] R. Bahlke, B. Moritz, and G. Snelting. A generator for language-specific debugging systems. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretative Techniques, SIGPLAN Notices 22(7)*, pages 92–101, July 1987.

[BOM81] B. Du Boulay, T. O'Shea, and J. Monk. The black box inside the glass box: presenting concepts to novices. *IJMMS*, 14:237–249, 1981.

[Bov87] J. D. Bovey. A debugger for a graphical workstation. *Software-practice and Experience*, 17(9):647–662, September 1987.

[BS86] R. Bahlke and G. Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transaction on Programming Languages and Systems*, 8(4):547–576, October 1986.

[CH87] Benjamin B. Chase and Robert T. Hood. Selective interpretation as a technique for debugging computationally intensive programs. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretative Techniques, SIGPLAN Notices 22(7)*, pages 113–124, July 1987.

[Chi89] C. Chin. A support tool for operational semantics. Undergraduate Project Report, LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, May 1989.

[CM86] L. M. Chirica and D. F. Martin. Toward compiler implementation correctness proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, April 1986.

[Coh78] Avra Cohn. High level proof in LCF. Internal Report CSR-35-78, Department of Computer Science, University of Edinburgh, November 1978.

[Des86] J. Despeyroux. Proof of translation in Natural Semantics. Research Report 514, INRIA, Sophia-Antipolis, France, April 1986.

[Des88] T. Despeyroux. TYPOL: a formalism to implement Natural Semantics. Technical Report 94, INRIA, Sophia-Antipolis, France, March 1988.

[DF87] Pierre Deransart and Gérard Ferrand. An operational formal definition of PROLOG. In *Proceedings of the Symposium on Logic Programming, San Francisco, California*, pages 162–172. IEEE Press, August 1987.

[dS92] Fabio Q. B. da Silva. *Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics.* Thesis submitted for the degree of Ph.D., LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, 1992.

[GB85] M. E. Garcia and W. J. Berman. An approach to concurrent systems debugging. In *Proceedings Fifth International Conference Distributed Computing Systems, Denver, CO*, pages 507–514, May 1985.

[GGM76] V. Giarratana, F. Gimona, and U. Montanari. Observability concepts in abstract data type specification. In *Proceedings 5th Symposium on Mathematical Foundations of Computer Science, Gdansk.* Springer-Verlag, 1976. Lecture Notes in Computer Science, 45.

[GM86] Narain Gehani and Andrew McGettrick, editors. *Software Specification Techniques.* International Computer Science Series. Addision-Weslley Publishing Company, 1986.

[Gri71] David Gries. *Compiler Construction for Digital Computers.* Wiley International Editions. John Wiley and Sons, 1971.

[Hen82] John Hennessy. Symbolic debugging of optimized code. *ACM Transaction on Programming Languages and Systems*, 4(3):324–344, July 1982.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[Jon77] C. Jones. Program quality and programmer productivity. Technical Report IBM TR 02.764, IBM, Santa Teresa Laboratory, San Jose, CA, 1977.

[Joy89] Jeffrey J. Joyce. A verified compiler for a verified microprocessor. Technical Report 167, Computer Laboratory, University of Cambridge, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, England, March 1989.

[Kah88] G. Kahn. Natural Semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258. North-Holland Publishing Company, 1988.

[KHC91] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring Semantics: a formal framework for specifying, implementing and reasoning about execution monitors. In *Proceedings ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada*, pages 338–352, June, 26–28 1991.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N. J., 1978.

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

[Lau79] Søren Lausen. Debugging techniques. *Software-practice and experience*, 9:51–63, 1979.

[Lew82] T. G. Lewis. *Software Engineering: Analysis and Verification.* Reston Publishing Company Inc., 1982.

[LL89] Beatrice Lazzerini and Lanfranco Lopriore. Abstraction mechanism for event control in program debugging. *IEEE Transactions on Software Engineering*, 15(7):890–901, July 1989.

[Mic] Sun Microsystems. Dbx - source level debugger. Manual pages, Sun Release 4.1.

[Mil71] Robin Milner. An algebraic definition of simulation between programs. In *Second International Joint Conference on Artificial Intelligence*, pages 481–489, London, 1971. The British Computer Society.

[Mil76] Robin Milner. Program semantics and mechanised proof. In *Foundations of Computer Science II*, pages 3–44. Math. Centre Amsterdam Tracts 82, 1976.

[Mil89] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[Moh88] Thomas G. Moher. PROVIDE: a process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, June 1988.

[Mor73] F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Proceedings SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, October 1973.

[Mos79] P. D. Mosses. A construtive approach to compiler correctness. Technical report, Aarhus University, 1979.

[MP67] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In J. Schwartz, editor, *Proceedings Symposium on Applied Mathematics*, pages 33–41. American Mathematical Society, 1967.

[MW72] Robin Milner and R. Weyhauch. Proving compiler correctness in a mechanised logic. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence*, chapter 3, pages 51–70. Edinburgh University Press, 1972.

[NO88] P. Nivela and F. Orejas. Initial behaviour semantics for algebraic specifications. In *Recent Trends in Data Type Specification, Selected Papers from the 5th Workshop on Specification of Abstract Data Types, Gullane, Scotland*, pages 184–207. Springer-Verlag, 1988. Lecture Notes in Computer Science, 332.

[Nor82] D. A. Norman. Some observations on mental models. In A. Stevens, editor, *Mental Models*, pages 7–19. Erlbaum, 1982.

[Par81] D. M. R. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science, 5th GI-Conference, Karlsuhe*, pages 167–183. Springer-Verlag, March 1981. Lecture Notes in Computer Science 104.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

[Pol81] Wolfgang H. Polak. *Compiler Specification and Verification*, volume 124 of *Lecture Notes in Computer Science*. Springer Verlag, 1981.

[PP92] Thomas Pittman and James Peters. *The Art of Compiler Design: Theory and Practice*. Prentice-Hall, Inc., 1992.

[Rei81] H. Reichel. Behavioural equivalence: a unifying concept for initial and final specification methods. In *Proceedings 3rd Hungarian Computer Science Conference*, pages 27–39, 1981.

[Sch87] Oliver Schoett. *Data Abstraction and the Correctness of Modular Programming*. PhD thesis, LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, February 1987.

[Sch90] Oliver Schoett. Behavioural correctness of data representation. *Science of Computer Programming*, 14:43–57, 1990.

[Sev87] Rudolph E. Seviora. Knowledge-based program debugging systems. *IEEE Software*, 4(3):20–32, 1987 1987.

[Sho83] M. L. Shooman. *Software Engineering.* New York: McGraw-Hill, 1983.

[Sim90] Todd Simpson. Correctness of a compiler specification for the SECD machine. Research Report 90/410/34, Department of Computer Science, University of Calgary, 2500 University Drive N.W., Calgary, Alberta, Canada, T2N 1N4, October 1990.

[ST87] D. Sannella and A. Tarleck. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34:150–178, 1987.

[Sto89] Joseph E. Stoy. *Denotational Semantics: the Scott-Strachey approach to programming language theory.* The MIT Press series in Computer Science. The MIT Press, 1989.

[SYO87] S. K. Skedzielewski, R. K. Yates, and R. R. Oldehoeft. DI: an interactive debugging interpreter for applicative languages. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretative Techniques, SIGPLAN Notices 22(7)*, pages 102–112, July 1987.

[TS85] Jean-Paul Tremblay and Paul G. Sorenson. *The Theory and Practice of Compiler Writing.* Computer Science Series. McGraw-Hill International Editions, 1985.

[TWW81] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. More on advice on structuring compilers and proving them correct. *Theorectical Computer Science*, 15:223–249, 1981.

[vdLW85] Frits van der Linden and Ian Wilson. An interactive debugging environment. *IEEE Micro*, 5(4):18–31, August 1985.

[WF91] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report Rice COMP TR91-160, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251-1892, April 1991.

[Zel84] Polle Trescott Zellweger. Interactive source-level debugging of optimized programs. Technical Report CSL-84-5,[P84-00047], Xerox PARC, Palo Alto, California, May 1984.

[ZJ91] Lawrence W. Zurawski and Ralph E. Johnson. Debugging optimized code with expected behaviour. University of Illinois at Urbana-Champaign., August 1991.