Formally Defining Debuggers: A Comparison of Three Approaches

Karen L. Bernstein, Eugene W. Stark

Department of Computer Science

State University of New York at Stony Brook

Stony Brook, NY 11794-4400 USA*

Abstract

Although there is a large body of literature on formal definitions of programming languages, relatively little work has been done in applying formal techniques to defining debuggers. Natural operational semantics, denotational semantics and transitional operational semantics are all proven techniques for formally defining programming languages. In this paper we present techniques for formally defining debuggers based on each of these three styles of definitions. We will investigate each style of definition by demonstrating how a simple debugger might be defined in each framework.

1 Introduction

According to a traditional view, a debugger is a tool that allows a programmer to get information about a program by observing the evaluation of the program according to the implementation. From a more general perspective, a debugger can be viewed as a tool that allows the programmer to gain additional insight into a program by observing the behavior of the program according to some well-defined operational model. This more general view of a debugger is attractive because it separates the issues related to describing a debugger from the issues related to its implementation. In this paper, we explore some of these issues related to defining a debugger according to an abstract operational model by comparing three possible approaches to formally defining debuggers.

One natural approach to defining a debugger is to extend the definition of the programming language to include the necessary debugging information. Each of the three definitions described in this paper takes a well established semantic formalism for programming languages and extends it in order to define a debugger. In his PhD dissertation, Amir Kishon used continuation-passing style denotational semantics as a framework for automatically generating debugging tools [Kis92]. Fabio da Silva, in his thesis, extended natural operational

*Research supported in part by NSF grants CCR-9320846 and CCR-8902215

semantics to establish a framework for proving compilers and debuggers correct [dS91]. In a previous work, the authors proposed transitional operational semantics as a formalism for designing novel debugging tools [BS95]. In order to compare the three approaches, we sketch how a simple debugger might be defined in each formalism.

Since we are primarily interested in comparing the semantic frameworks, we will define a very simple debugger for a very simple programming language. Definitions of more sophisticated debuggers are available in each of the original works [Kis92, dS91, BS95]. For our programming language, we use a strict functional language and for our debugger, we define a stepper that allows the user to specify the number of steps that the program should execute. For our analysis, we are interested in comparing how in each case: (1) the programming language is defined, (2) the debugger is defined and (3) the relationship between the programming language definition and the debugger definition.

For the purpose of comparing different semantic frameworks, we will consider a strict functional language. The abstract syntax of our language is:

$$k \in \text{Constants} \qquad a \in \text{Identifiers}$$

$$e \in \text{Expressions} ::= k \mid a \mid (\textbf{fn}\ a \Rightarrow e) \mid (e_1\ e_2)$$

This a simple language, where the only programming language constructs are function definition and application. Elements of the set of constants are denoted by the letter k. We can think of the set of constants to simply be the set of integers. Elements of the set of identifiers will be denoted by the letter a. Expressions in the programming language can take several forms and will be denoted by the letter e. Any constant or identifier is a valid expression. A function definition of the form $(\textbf{fn}\ a \Rightarrow e)$ is a valid expression and it denotes the function that maps each value v to the term $e[v/a]$ (e with v substituted for a). An application of the form $(e_1\ e_2)$ is a valid expression and it denotes the application of the expression e_1 to e_2.

The application $(e_1\ e_2)$ is only well-defined in the case that e_1 is a function definition and e_2 is fully evaluated (is either a constant or a function definition). This definition of application corresponds to call-by-value semantics. When the semantic framework allows us to specify the order of evaluation, we would like to specify left-to-right evaluation for applications. That is, first the operator e_1 evaluates fully, then the operand e_2 evaluates fully and finally the fully evaluated operator is applied to the fully evaluated operand.

In order to see how a simple expression in the programming language will evaluate, we will consider the expression, $((\textbf{fn}\ a \Rightarrow 1)\ ((\textbf{fn}\ b \Rightarrow b)\ 2))$. This simple expression is of the form, $(e_1\ (e_2\ e_3))$, where $e_1 = (\textbf{fn}\ a \Rightarrow 1)$, $e_2 = (\textbf{fn}\ b \Rightarrow b)$ and $e_3 = 2$. The expression $e_1 = (\textbf{fn}\ a \Rightarrow 1)$ corresponds to the constant function that always returns the value 1 and the expression e_2 corresponds to the identity function that always returns the value to which it is applied. Therefore this expression should evaluate in two steps to the final result 1.

$$((\textbf{fn}\ a \Rightarrow 1)\ ((\textbf{fn}\ b \Rightarrow b)\ 2)) \longrightarrow ((\textbf{fn}\ a \Rightarrow 1)\ 2) \longrightarrow 1$$

In the evaluation of this expression, first the identity function is applied to the value 2 yielding the value 2, then the constant function is applied to the value 2 yielding the value 1.

The outline of the paper is as follows. Section 2 defines the debugger in denotational semantics. Section 3 defines the debugger in natural operational semantics. Section 4 defines the debugger in transitional operational semantics. In order to compare the three formalisms, in each case we will first give a formal definition for our simple programming language and we will look at how a simple programming expression would evaluate according to the language definition. Next, we will define a simple debugger that allows the user to issue a command that the program being debugged should evaluate for some requested number of steps. Finally, for each formalism we will look at the properties of the definitions that are important in the framework. Section 4 concludes the paper by commenting on the issues that seem fundamental to defining debuggers.

2 Denotational Semantics

In denotational semantics, a correspondence is defined between expressions in the programming language and functions in a mathematical domain. Denotational semantics is a very elegant and concise way to define a programming language. The difficulty in defining debuggers in this framework is that there is no explicit notion of an evaluation step. In their work, Kishon, Hudak and Consel show how continuation-passing style denotational semantics can be viewed as introducing an explicit notion of an evaluation step into the definition of the programming language [KHC91]. Kishon *et al.* present a *monitoring semantics* as an extension to the continuation-passing denotational semantics for a language, where monitors are tools such as debuggers, profilers, and tracers that can view but not modify the execution of a program. Kishon *et al.* are then able to use partial evaluation techniques to automatically generate program execution monitors.

2.1 Programming language definition

In denotational semantics each expression in the programming language is mapped directly to its meaning (denotation) [Sch86]. For this programming language, the denotable values are constants, identifiers, functions and $\perp$, where $\perp$ is the value of an expression that never terminates. This mapping is done by means of a valuation function, $\mathcal{E}$, which returns the corresponding denotation for any programming language expression with respect to the current environment (a function from identifier names to denotable values). In continuation-passing denotational semantics, the valuation function has an additional argument, the continuation function. A continuation is a function that takes an intermediate value as an argument and returns the final result of the program. Intuitively, a continuation is the "remainder of the program."

We will use $[\,]$ to designate the environment that maps every identifier to $\perp$ and $\rho[a \mapsto k]$ to designate the environment that maps the identifier a to the denotable value k and otherwise behaves like ρ. Denotations are assigned to constants by means of a function $\mathcal{K}$, which maps each constant to its value in the domain. We will use **id** to designate the identity continuation: that is, the continuation function that returns whatever value is passed as an argument to the continuation.

The definition of our programming language is given in two parts. First we describe the *semantic algebras*, which define the semantic objects that are the targets of the valuation function.

Semantic Algebras:

$k \in$ *Constant-values* $=$ Int $+ \ldots$
$v \in$ *Denotable-values* $=$ *Constant-values* $+$ *Function-values*
$\rho \in$ *Environments* $=$ *Identifiers* $\rightarrow$ *Denotable-values*
$f \in$ *Function-values* $=$ *Denotable-values* $\rightarrow$ *Continuations* $\rightarrow$ *Denotable-values*
$\kappa \in$ *Continuations* $=$ *Denotable-values* $\rightarrow$ *Denotable-values*

The semantic algebras include a set of constant values that contains at least the set of the integers. The denotable values consist of the constant values and the function values. An environment is a function from identifier names to denotable values. In this framework, functions are slightly unusual. Rather than a function value simply being a function from denotable values to denotable values, instead a function takes an additional argument that corresponds to the continuation. Intuitively the continuation describes how to take the value output by the function and produce the final result of the whole program. Typically, continuations are used in denotational semantics to describe complicated control constructs; here they are used to obtain a notion of an evaluation step.

We can now define the valuation function ($\mathcal{E}$):

Valuation Function:

$\mathcal{E}$: *Expressions* $\rightarrow$ *Environments* $\rightarrow$ *Continuations* $\rightarrow$ *Denotable-values*
$\mathcal{E}\,[\![\, a \,]\!]\, \rho\ \kappa = \kappa(\rho(a))$
$\mathcal{E}\,[\![\, k \,]\!]\, \rho\ \kappa = \kappa(\mathcal{K}(k))$
$\mathcal{E}\,[\![\, (\mathbf{fn}\ a => e) \,]\!]\, \rho\ \kappa = \kappa(\lambda x.(\mathcal{E}\,[\![\, e \,]\!]\, \rho[a \mapsto x]))$
$\mathcal{E}\,[\![\, e_1\ e_2 \,]\!]\, \rho\ \kappa = \mathcal{E}\,[\![\, e_1 \,]\!]\, \rho\ \{\lambda x_1.\mathcal{E}\,[\![\, e_2 \,]\!]\, \rho\ \{\lambda x_2.(x_1\ x_2)\kappa\}\}$

The valuation function takes three arguments (a programming language expression, an environment and a continuation function) and returns the corresponding denotation. The first rule says, to evaluate the identifier a in environment ρ with continuation κ, first apply the environment function ρ to the identifier a to determine the value of a in the environment ρ and then apply the continuation κ to the result to get the final value. Similarly, the second rule says, to evaluate the constant k in environment ρ with continuation κ, first apply $\mathcal{K}$ to the constant to determine its value and then apply the continuation κ to the result to get the final value. The third rule says to evaluate a function definition of the form ($\mathbf{fn}\ a => e$) in environment ρ with continuation κ, apply the continuation κ to the function value that accepts an argument x and returns the value denoted (according to the valuation function $\mathcal{E}$) by the expression e in environment $\rho[x \mapsto a]$. The last rule says to find the meaning of an application of the form $(e_1\ e_2)$ in environment ρ with continuation κ, first determine the function value denoted by the expression e_1 in environment ρ, then determine the argument value denoted by the the expression e_2 in environment ρ, apply the function to its argument and then finally apply the continuation κ to the result. Notice that since we want to define left-to-right evaluation, we evaluate the operator of the application before we evaluate the operand.

We can view the continuation as a stack of functions that indicates what remains to be done in the evaluation of the programming language expression. The series of transitions that demonstrates the evaluation of our example programming language expression are given in figure 1. In step 1, the evaluation starts with our programming language expression in the empty environment with identity continuation (no other expressions are currently on the stack). In step 2, the application is expanded. The current expression (the top expression on the stack) is now the operator for the application; the second function on the stack corresponds to the operand for the application; and the third function on the stack corresponds to the actual application of the operator to the operand. In step 3, the function definition (**fn** a => 1) is transformed into the function it denotes. In step 4, the result is popped off the stack and substituted for x_2 in the expressions in the stack. In step 5, the current application is expanded, adding new expressions to the stack. The evaluation continues in this manner until step 14, where the final result, 1, is determined.

2.2 Debugger definition

In order to define the debugger, the syntax of the program is "tagged" with annotations to indicate points of interest to the debugger. This information would usually be supplied by the programming environment. For our sample programming language expression, the tags are indicated below. In this case, the only tag is {app} and it indicates the location of an application.

$$(\{\text{app}\}(\textbf{fn}\ a \Rightarrow 1)\ (\{\text{app}\}(\textbf{fn}\ b \Rightarrow b)\ 2))$$

In this approach the debugger is defined by introducing a *monitoring state* and a *monitoring function*. The monitoring state contains the additional state information necessary for defining the debugger. The monitoring function defines how the monitoring state should be updated as the evaluation progresses. The monitoring function takes four parameters (the current annotation, the current expression, the current environment and the current monitoring state) and returns the new monitoring state. Since our example debugger simply counts steps, we only need to store a natural number in the monitoring state. The monitoring function is defined as follows.

Monitoring function:
$\mathcal{M}^{\mathcal{E}}$: *Annotations* $\rightarrow$ *Expressions* $\rightarrow$ *Environments* $\rightarrow$ *Monitoring States* $\rightarrow$ *Monitoring States*
$\mathcal{M}^{\mathcal{E}}\ [\text{app}]\ [\![\, e \,]\!]\ \rho\ n = n + 1$

	Expression
step 1	$\mathcal{E}\,[\![((\textbf{fn}\ a => 1)\ ((\textbf{fn}\ b => b)\ 2))]\!]\,[\,]$
step 2	$\mathcal{E}\,[\![(\textbf{fn}\ a => 1)]\!]\,[\,]$ $\lambda x_2.\mathcal{E}\,[\![((\textbf{fn}\ b => b)\ 2)]\!]\,[\,]$ $\lambda x_1.(x_2\ x_1)$
step 3	$\lambda x.\mathcal{E}\,[\![1]\!]\,[a \mapsto x]$ $\lambda x_2.\mathcal{E}\,[\![((\textbf{fn}\ b => b)\ 2)]\!]\,[\,]$ $\lambda x_1.(x_2\ x_1)$
step 4	$\mathcal{E}\,[\![((\textbf{fn}\ b => b)\ 2)]\!]\,[\,]$ $\lambda x_1.((\lambda x.\mathcal{E}\,[\![1]\!]\,[a \mapsto x])\ x_1)$
step 5	$\mathcal{E}\,[\![(\textbf{fn}\ b => b)]\!]\,[\,]$ $\lambda x_3.\mathcal{E}\,[\![2]\!]\,[\,]$ $\lambda x_2.(x_3\ x_2)$ $\lambda x_1.((\lambda x.\mathcal{E}\,[\![1]\!]\,[a \mapsto x])\ x_1)$
step 6	$\lambda x.\mathcal{E}\,[\![b]\!]\,[b \mapsto x]$ $\lambda x_3.\mathcal{E}\,[\![2]\!]\,[\,]$ $\lambda x_2.(x_3\ x_2)$ $\lambda x_1.((\lambda x.\mathcal{E}\,[\![1]\!]\,[a \mapsto x])\ x_1)$
step 7	$\mathcal{E}\,[\![2]\!]\,[\,]$ $\lambda x_2.((\lambda x.\mathcal{E}\,[\![b]\!]\,[b \mapsto x])\ x_2)$ $\lambda x_1.((\lambda x.\mathcal{E}\,[\![1]\!]\,[a \mapsto x])\ x_1)$
step 8	2 $\lambda x_2.((\lambda x.\mathcal{E}\,[\![b]\!]\,[b \mapsto x])\ x_2)$ $\lambda x_1.((\lambda x.\mathcal{E}\,[\![1]\!]\,[a \mapsto x])\ x_1)$
step 9	$((\lambda x.\mathcal{E}\,[\![b]\!]\,[b \mapsto x])\ 2)$ $\lambda x_1.((\lambda x.\mathcal{E}\,[\![1]\!]\,[a \mapsto x])\ x_1)$
step 10	$\mathcal{E}\,[\![b]\!]\,[b \mapsto 2]$ $\lambda x_1.((\lambda x.\mathcal{E}\,[\![1]\!]\,[a \mapsto x])\ x_1)$
step 11	2 $\lambda x_1.((\lambda x.\mathcal{E}\,[\![1]\!]\,[a \mapsto x])\ x_1)$
step 12	$((\lambda x.\mathcal{E}\,[\![1]\!]\,[a \mapsto x])\ 2)$
step 13	$\mathcal{E}\,[\![1]\!]\,[a \mapsto 2]$
step 14	1

Figure 1: Evaluation sequence for definition in continuation-passing denotational semantics

The definition simply says, if the current evaluation step is an application (is annotated with “app”) then increment the counter in the monitoring state.

2.3 Properties of definitions

The denotational style of definition has several very nice properties. Foremost, this style of definition in conjunction with partial evaluation techniques allows debugging tools to be automatically generated. This style of definition is also widely used in programming language semantics, so existing programming language definitions can be used to generate debuggers. In addition, the formalism allows a variety of monitoring semantics to be defined by parameterizing language specifications with respect to monitor specifications.

Two key properties of the debugger are guaranteed with style of definition: (1) the monitoring semantics cannot change the semantics of original programming language, and (2) the monitoring semantics are compositional. Both of these properties seem important to debugger definitions.

We see in the sample evaluation sequence that the formalism introduces extra evaluation steps that are not of interest to the programmer. This is compensated for by means of the program annotations, but for a more complicated debugger definition the exact effect of the program annotations could be difficult to understand.

3 Natural Operational Semantics

Natural operational semantics uses proof rules to define the values to which programming language expressions evaluate. Natural semantics is widely used for defining programming languages, because definitions in this form can be very intuitive and concise. In addition, it provides a framework for proving interesting properties of the programming language and as well as properties of programs written in the language. In his dissertation, da Silva uses a variant of natural semantics to formally define debuggers. His principal goal was to construct a framework for proving debuggers correct. His framework allows the formal specification of a debugger and provides methods for proving the implementation correct with respect to the specification.

3.1 Programming language definition

In natural operational semantics a programming language is defined by inference rules of the form:

$$\frac{e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n}{e \Downarrow v}$$

The expressions on top of the bar are the premises and the expression on the bottom is the conclusion. A rule of this form can be read, “if the expression e_1 evaluates to the value v_1 and ... and the expression e_n evaluates to the value v_n then the expression e evaluates to the value v.” The following three such rules define our programming language:

$$k \Downarrow k \;\; (\mathrm{nat1}) \qquad\qquad (\mathbf{fn}\ a => e) \Downarrow (\mathbf{fn}\ a => e) \;\; (\mathrm{nat2})$$

$$\frac{e_1 \Downarrow (\mathbf{fn}\ a => e_1') \quad e_2 \Downarrow v_2 \quad e_1'[v_2/a] \Downarrow v}{(e_1\ e_2) \Downarrow v} \;\; (\mathrm{nat3})$$

The first two rules say that constants and function definitions all evaluate to themselves; that is, they are fully evaluated. The last rule says, if the expression e_1 evaluates to the function definition ($\mathbf{fn}\ a => e_1'$) and the expression e_2 evaluates to the value v_2 and the expression e_1' with v_2 substituted for a evaluates to the value v, then the application with the expression e_1 applied to the expression e_2 evaluates to the value v.

The proof tree below shows how the given rules can be used to prove that the function ($\mathbf{fn}\ a => 1$) (($\mathbf{fn}\ b => b$) 2) evaluates to the value 1.

$$\dfrac{\dfrac{(\mathrm{nat2})}{(\mathbf{fn}\ a => 1) \Downarrow (\mathbf{fn}\ a => 1)} \quad \dfrac{\dfrac{(\mathrm{nat2})}{(\mathbf{fn}\ b => b) \Downarrow (\mathbf{fn}\ b => b)} \quad \dfrac{(\mathrm{nat1})}{2 \Downarrow 2} \quad \dfrac{(\mathrm{nat1})}{b[2/b] \Downarrow 2}}{(\mathbf{fn}\ b => b)\ (2) \Downarrow 2} \quad \dfrac{(\mathrm{nat1})}{1[2/a] \Downarrow 1}}{(\mathbf{fn}\ a => 1)\ ((\mathbf{fn}\ b => b)\ (2)) \Downarrow 1}$$

In order to be able to define our debugger with respect to this definition, we need some way to extract a notion of an evaluation step. Da Silva introduces the notion of an evaluation step by proposing an alternate reading of the rules of the definition. He treats the proof rules as rewrite rules that specify the evaluation sequence. In da Silva's treatment, the rule (nat3) can be read, "In order to evaluate the expression $e1$ applied to expression $e2$: (1) first evaluate e_1 to an expression of the form ($\mathbf{fn}\ a => e_1'$), (2) next evaluate e_2 and call the result v_2, (3) finally evaluate the expression e_1' with v_2 substituted for a." With this interpretation of the rules, the evaluation state can be written as a stack of evaluation sequences that are yet to be performed. The proof rules are used to replace the expressions on the stack by their premises.

The evaluation sequence for our example programming language expression is shown in figure 2. In step 1, at the start of the evaluation, our goal is to find the value v to which the expression (($\mathbf{fn}\ a => 1$) (($\mathbf{fn}\ b => b$) 2)) evaluates. In step 2, we expand the application according to the proof rule for application. In this case, if ($\mathbf{fn}\ a => 1$) evaluates to ($\mathbf{fn}\ a => e_1$) and (($\mathbf{fn}\ b => b$) 2) evaluates to v_1 then e_1 with v_1 substituted for a evaluates to v. In step 3, we pop ($\mathbf{fn}\ a => 1$) $\Downarrow$ ($\mathbf{fn}\ a => e_1$) off the stack and substitute 1 for e_1 in the remaining goals. The remaining evaluation steps proceed in a similar manner. Each evaluation step either: (1) puts new terms on the stack by rewriting according to the proof rules or, (2) substitutes values for variable names based on the values derived by applying axioms. In the last evaluation step, we have evaluated all the goals and all that remains is the result that v does in fact evaluate to the final result 1.

The method described here for treating the proof rules as rewrite rules only works if the proof rules are in certain formats. Da Silva defines a restricted form of proof rules which he calls *rational semantics* for which this technique works.

step 1	(**fn** a => 1) ((**fn** b => b) 2)	$\Downarrow$	v
step 2	(**fn** a => 1)	$\Downarrow$	(**fn** a => e_1)
	((**fn** b => b) 2)	$\Downarrow$	v_1
	$e_1[v_1/a]$	$\Downarrow$	v
step 3	((**fn** b => b) 2)	$\Downarrow$	v_1
	$1[v_1/a]$	$\Downarrow$	v
step 4	(**fn** b => b)	$\Downarrow$	(**fn** b => e_2)
	2	$\Downarrow$	v_2
	$e_2[v_2/b]$	$\Downarrow$	v_1
	$1[v_1/a]$	$\Downarrow$	v
step 5	2	$\Downarrow$	v_2
	$b[v_2/b]$	$\Downarrow$	v_1
	$1[v_1/a]$	$\Downarrow$	v
step 6	$b[2/b]$	$\Downarrow$	v_1
	$1[v_1/a]$	$\Downarrow$	v
step 7	$1[2/a]$	$\Downarrow$	v

Figure 2: Evaluation sequence for definition in natural semantics

3.2 Debugger definition

In order to define the debugger, da Silva introduces the notion of an *evaluation history*, which is sequence of past evaluation states. In addition, the program state information is augmented with information about the state of the debugger. The debugger is specified using a functional language that acts upon the evaluation histories and debugging states.

One complication with the evaluation sequence as defined by da Silva's method is that not all of the evaluation steps are of interest to us; some steps are simply an artifact of the rewrite rules. For example, we are not really interested in steps 2 and 4 in figure 2, since they simply correspond to the expansion of the application. In order to get around this issue, da Silva provides a facility for defining predicates that allows the debugger definition to skip the steps that are not really of interest. We will assume the definition of the predicate "noapp" which is true whenever the next evaluation step in the evaluation history is not the expansion of an application. The debugger can now be defined as follows:

fun step (h, n) = if $n > 0$ then case next(h) of
  h1 $\Rightarrow$ if noapp$(h1, st)$ then step$(h1, n-1)$ else step$(h1, n)$

The debugger definition above recursively defines the the debugger command step(h, n) as follows. If there are still more evaluation steps to be done ($n > 0$) then do the following. If the current evaluation step is not the expansion of an application, evaluate for another $n-1$ steps (call step$(h1, n-1)$), otherwise evaluate for another n steps (call step$(h1, n)$).

3.3 Properties of definitions

Da Silva identified several important requirements for a debugger: (1) a debugger must be robust, that is all debugging commands must evaluate to some result at every debugging state; (2) at every state there must be a debugging command that advances the evaluation; and (3) the debugging states must be consistent with the evaluation states according to the semantics of the programming language.

An important issue in proving debuggers correct is finding a way to decide when two debuggers are equivalent. The evaluation steps that are introduced as an artifact of the semantic framework are an impediment to determining whether two programs are equivalent. Da Silva gets around this by defining a signature that determines which evaluation steps are visible and which should be disregarded. Two debugger specifications are then equivalent if exhibit equivalent behavior with respect to "visible" evaluation steps.

It seems that there are several aspects to this semantic formalism that could make it hard to formally prove properties. First, the programming language for defining the debuggers is relatively sophisticated and it would require quite a bit of effort to prove properties of specifications written in the language. Second, the extra evaluation steps introduced by the application of the rewriting rules, introduce quite a bit of complexity into the formalism. This is an issue both in defining the debugger and in defining the equivalence between debuggers.

4 Transitional Operational Semantics

In transitional operational semantics the evaluation of a programming language expression is defined in terms of explicit incremental evaluation steps. This formalism has been used widely for describing interactive systems [Mil89], but has been less popular for defining sequential programming languages. In a previous paper we showed how this semantic formalism could be used to define novel debuggers [BS95]. We chose transitional operational semantics because it provides an intuitive notion of an evaluation step and it is convenient for describing the interaction between both the user and the debugger and the debugger and the programming language.

4.1 Programming language definition

In the definition of the programming language the unlabeled transitions correspond to our usual notion of an evaluation step and the labeled transitions provide other necessary information. The definition for the unlabeled transitions is shown below.

The first rule says that if the expression e_1 can do an evaluation step and become the expression e_1', then the application $(e_1\ e_2)$ can take an evaluation step and become the application $(e_1'\ e_2)$. The second rule says if the expression e_2 can take an evaluation step and become the expression e_2', then the application $((\mathbf{fn}\ a => e_1)\ e_2)$ can take an evaluation step and become the application $((\mathbf{fn}\ a => e_1)\ e_2')$. Notice that in the second rule, the operator of the application is required to be of the form $(\mathbf{fn}\ a => e)$. Thus, these rules define left-to-right evaluation, since the operator must evaluate completely before the operand can evaluate. The third rule says, if e with the value v substituted for a is e' then the function definition ($\mathbf{fn}\ a \Rightarrow e$) applied the value v can take an evaluation step and become e'.

$$\frac{e_1 \longrightarrow e_1'}{(e_1\ e_2) \longrightarrow (e_1'\ e_2)}\ \text{(tr1)} \qquad \frac{e_2 \longrightarrow e_2'}{((\mathbf{fn}\ a => e_1)\ e_2) \longrightarrow ((\mathbf{fn}\ a => e_1)\ e_2')}\ \text{(tr2)}$$

$$\frac{e \xrightarrow{[v/a]} e'}{((\mathbf{fn}\ a => e)\ v) \longrightarrow e'}\ \text{(tr3)}$$

In addition to the evaluation rules above, our definition includes the following rules for defining the valid substitution transitions (transitions labeled with $[e/a]$).

$$k \xrightarrow{[e/a]} k\ \text{(sub1)} \qquad a \xrightarrow{[e/a]} e\ \text{(sub2)} \qquad \frac{b \neq a}{b \xrightarrow{[e/a]} b}\ \text{(sub3)}$$

$$\frac{e_1 \xrightarrow{[e/a]} e_1' \quad e_2 \xrightarrow{[e/a]} e_2'}{e_1\ e_2 \xrightarrow{[e/a]} e_1'\ e_2'}\ \text{(sub4)} \qquad \frac{e_2 \xrightarrow{[e/a]} e_2'}{(\mathbf{fn}\ b => e_2) \xrightarrow{[e/a]} (\mathbf{fn}\ b => e_2')}\ \text{(sub5)}$$

$$(\mathbf{fn}\ a => e_2) \xrightarrow{[e/a]} (\mathbf{fn}\ a => e_2)\ \text{(sub6)}$$

The first rule says, the constant k with e substituted for a is k. The second rule says, the identifier a with e substituted for a is e. The remaining rules can be read in a similar manner.

Using this programming language definition, the proof trees for the two evaluation steps necessary to evaluate our sample programming expression are given in figure 3. Notice that in this formalism, no extra evaluation steps are introduced. In the first evaluation step, ($\mathbf{fn}\ a => a$) is applied to 2, yielding the value 2. In the second evaluation step, ($\mathbf{fn}\ a => 1$) is applied to 2 yielding 1.

4.2 Debugger definition

For our transition-style semantics, the debugging state $\langle e, n\rangle$ contains two parts: e, the current programming language state (a programming language expression) and n, a counter. We also introduce the predecessor function *pred*. The abstract syntax for our debugging language is as follows:

$$n \in \text{Natural Numbers} \qquad e \in \text{Expression} \qquad d \in \text{Debugging States} ::= \ \langle e, n\rangle \mid \text{pred}(n)$$

In our formalism, we define the debugger as a set of rules "on top" of the programming language definition. We distinguish transitions in the debugging language from transitions in the programming language by using a double arrow in the debugging transitions.

$$\dfrac{\dfrac{\dfrac{}{a \xrightarrow{[2/a]} 2}\ (\mathrm{sub2})}{((\mathbf{fn}\ a => a)\ 2) \longrightarrow 2}\ (\mathrm{tr3})}{((\mathbf{fn}\ a => 1)\ ((\mathbf{fn}\ a => a)\ 2)) \longrightarrow ((\mathbf{fn}\ a => 1)\ 2)}\ (\mathrm{tr2})$$

$$\dfrac{\dfrac{}{1 \xrightarrow{[2/a]} 1}\ (\mathrm{sub1})}{((\mathbf{fn}\ a => 1)\ 2) \longrightarrow 1}\ (\mathrm{tr3})$$

Figure 3: Evaluation sequence for definition in transitional operational semantics

$$\mathrm{pred}(n) \Longrightarrow n-1 \ (\text{where } n>0) \qquad \dfrac{e \longrightarrow e' \quad \mathrm{pred}(n) \Longrightarrow n'}{\langle e, n\rangle \Longrightarrow \langle e', n'\rangle}$$

The first rule actually defines an infinite set of rules. It says every for every natural number n greater than zero, pred(n) can do a transition and become $n-1$. The second rule says that if the programming language expression e can do an evaluation step and become e' and the counter pred(n) can do an evaluation step and become n' then the debugging expression $\langle e, n\rangle$ can do an evaluation step and become $\langle e', n'\rangle$. If the initial debugging state is $\langle e, n\rangle$, then the expression will evaluate for exactly n steps yielding the debugging state $\langle e', 0\rangle$, where e' corresponds exactly to the expression that results when e evaluates for n steps according to the programming language definition.

This debugger definition is attractive in that is very simple. It captures our intuitive notion of how a debugger could interact with the execution of a program. As we show in [BS95] this formalism also allows us to prove some desirable properties of the debugger.

4.3 Properties of definitions

Rules defined in our formalism have several nice properties, (1) the debugger definition does not change the meaning of the original programming language, (2) the debugging states have well-defined interaction with the programming language, and (3) there is an intuitive and well-defined notion of an evaluation step.

The main difficulty in defining the debugger in this style is that traditionally, transition-style semantics has not been popular for defining sequential programming languages. As such, there are not many existing definitions in this format and the issues in proving properties of definitions in this format are less well understood. However, other researchers have recently also started using this formalism, especially for dealing with issues that introduce interaction into a programming language [Gor95, Jef95]. As this research develops, it should become easier to construct the necessary programming language definitions.

The other issue is that it still remains to see how our definition can be used as the basis of an implementation. Recent research has shown how source code transformations can be used effectively to implement debuggers [TA90, Spa94]. We are currently working on using this implementation technique to develop an implementation based on our style of formal definition.

5 Conclusions

In this paper we compared three different approaches to defining a simple debugger for a functional programming language. From these definitions, we can see some of issues that seem fundamental to defining debuggers. All three definitions had two parts: first an abstract operational model for the programming language was defined; then the debugger was defined with respect to that operational model. In order for a debugger to be well-defined it seems necessary to at least consider the following three issues:

- the interaction between the debugger and the user,
- the operational behavior of the debugger itself,
- and the interaction between the debugger and the programming language.

Even though our example debugger in this paper was very simple, it seems that some of these issues are important for any formal definition of a debugger.

The three approaches were different in the properties that were identified as important for the debugger. Some of the properties that seem pertinent to any debugger are: (1) every debugging command must evaluate to some result at every debugging state; (2) there should be a well defined relationship between the debugging and programming language states; (3) the debugger definition should not change the semantics of the original programming language; and (4) debugger definitions should be compositional. Some properties seemed to only pertain to specific formalisms. In particular, the approaches based on natural operational and denotational semantics introduced extraneous evaluation steps that complicated the definition of the debugger.

There are many more interesting issues related to formally defining debuggers than are even touched in this paper. There are of course many other important programming paradigms and much more interesting debuggers. In particular, a significant amount of work has been done in debuggers for imperative languages, logic programming languages [Duc94] and parallel programming languages. Especially interesting debugging tools include automated and algorithmic debuggers [Sha83, Duc93], and most closely related to this paper, algorithmic debuggers for functional programming languages [NF94].

This research area does seem to have some promising applications. High-level languages such as Haskell, SML and Prolog each have a formal definition at its foundations so it seems reasonable that a debugger for such a language should have a formal definition that is closely related to the definition of the language itself.

References

[BS95] Karen L. Bernstein and Eugene W. Stark. Operational semantics of a focusing debugger. In *Eleventh Conference on the Mathematical Foundations of Programming Semantics*, New Orleans, USA, March 1995.

[dS91] Fabio Q.B. da Silva. *Correctness Proofs of Compilers and Debuggers: an Approach Based on Structured Operational Semantics.* PhD thesis, University of Edinburgh,Edinburgh, Scotland, October 1991. LFCS, Department of Computer Science.

[Duc93] Mireille Ducassé. A pragmatic survey of automated debugging. In *1st International Workshop on Automated and Algorithmic Debugging*, pages 1–15. Department of Computer and Information Science, Linköping University, May 1993.

[Duc94] Mireille Ducassé. Logic programming environments: Dynamic program analysis and debugging. *Journal of Logic Programming*, 19,20:351–384, 1994.

[Gor95] Andrew Gordon. Bisimilarity as a theory of functional programming. In *Proceedings of Mathematical Foundations of Programming Semantics.* Elsevier Science, 1995.

[Jef95] Alan Jeffrey. A fully abstract semantics for a nondeterministic functional language with monadic types. In *Proceedings of Mathematical Foundations of Programming Semantics.* Elsevier Science, 1995.

[KHC91] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 338–352. ACM Press, June 1991.

[Kis92] Amir Shai Kishon. *Theory and Art of Semantics-Directed Program Execution Monitoring.* PhD thesis, Yale University, May 1992.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall International, Berlin, 1989.

[NF94] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–369, July 1994.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* The MIT Press, Boston, Massachusetts, 1986.

[Sha83] Ehud Y. Shapiro. *Algorithmic Program Debugging.* ACM Distinguished Dissertations. The MIT Press, Cambridge, MA, 1983.

[Spa94] Jan Sparud. An embryo to a debugger for Haskell. Technical report, Chalmers University of Technology, Göteborg, Sweden, 1994. FTP://ftp.cs.chalmers.se:/pub/cs-reports/papers/hsdbg.ps.gz.

[TA90] A. P. Tolmach and A. W. Appel. Debugging Standard ML without reverse engineering. In *1990 ACM Conference on Lisp and Functional Programming*. Association for Computing Machinery, ACM Press, June 1990.