Electronic Notes in Theoretical Computer Science 1 (1995)
URL: http://www.elsevier.nl/locate/entcs/volume1.html 17 pages

Operational Semantics of a Focusing Debugger

Karen L. Bernstein and Eugene W. Stark [1]

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400 USA

Abstract

This paper explores two main ideas: (1) a debugger for a programming language ought to have a formal semantic definition that is closely allied to the formal definition of the language itself; and (2) a debugger for very high level programming language ought to provide support for exposing hidden information in a controlled fashion. We investigate these ideas by giving formal semantic definitions for a simple functional programming language and an associated debugger for the language. The formal definitions are accomplished using structured operational semantics, and they demonstrate one way in which the formal definition of a debugger might be built "on top of" the formal definition of the underlying language. The debugger itself provides the novel capability of allowing the programmer to "focus" or shift the scope of attention in a syntax-directed fashion to a specific subexpression within the program, and to view the execution of the program from that vantage. The main formal result about the debugger is that "focusing preserves meaning," in the sense that a program being debugged exhibits equivalent (bisimilar) operational behavior regardless of the subexpression to which the focus has been shifted.

1 Introduction

This paper explores two main ideas. First, in order to provide a clear definition of the relationship between a programming language and its debugger, the debugger ought to have a formal semantic definition that is closely allied to the formal definition of the language itself. For example, if a programming language is defined in terms of a rewriting semantics but has a stack-based implementation, the debugger should be defined in terms of the formal definition rather than the implementation. The second main idea is that, whereas much has been learned in the past few decades about how programming languages can be designed to help manage complexity by facilitating and enforcing information hiding, the design of debuggers has in general not kept pace with these advances. In particular, a debugger for a very high level programming

[1] Research supported in part by NSF grants CCR-9320846 and CCR-8902215

language that strongly enforces information hiding ought to provide some sort of support for exposing hidden information in a controlled fashion.

We investigate the above two themes by giving formal semantic definitions for a simple functional programming language and an associated debugger for that language. The formal definitions are accomplished using structured operational semantics [13], and they demonstrate one way in which the formal definition of a debugger might be built "on top of" the formal definition of the underlying language. The debugger itself provides the novel capability of allowing the programmer to "focus" or shift the scope of attention in a syntax-directed fashion to a specific subexpression within the program, and to view the execution of the program from that vantage. For example, one might shift one's focus of attention inside the scope of a block, thereby obtaining access to a binding environment that would be hidden at the "top level." Our main formal result about the debugger is that "focusing preserves meaning," in the sense that a program being debugged exhibits equivalent (bisimilar) operational behavior regardless of the subexpression to which the focus has been shifted.

An example of a modern very high level programming language to which our ideas might be applied is Standard ML [11]. Compilers for very high level programming languages like Standard ML typically transform the source code rather dramatically before object code is produced, thus increasing the discrepancy between the conceptual model used by the programmer when writing the code and the implementation model actually used when the program is executing. For example, the continuation-passing transformations applied by the Standard ML of New Jersey (SML/NJ) compiler [2] can result in object code that bears little resemblance to the original source. Instead of trying to track the relationship between radically different source and object code, the experimental debugger shipped with SML/NJ [15] works by instrumenting the source code and capturing at run time the information necessary to present to the programmer a traditional stack-based run-time environment. In essence, the SML/NJ debugger creates and presents to the programmer a virtual implementation model that does not have very much to do with the implementation model actually used by the compiler.

The method used by the SML/NJ debugger raises an interesting question. If a debugger is to present the programmer with a virtual implementation model, what should that implementation model be? The debugger for SML/NJ reduces the mental burden on the programmer by presenting the same conceptual model as was used when writing the code in the first place. Another possible approach is based on the idea that a debugger can provide an alternative implementation model, perhaps a model that is drastically different from the underlying programming language definition. Such an alternative perspective could provide additional insight and assist the programmer in finding bugs. For example, even if a programming language is defined in terms of a stack-based semantics and has a stack-based implementation, a debugger defined in terms of a rewriting semantics could be a helpful tool. In developing such an alternative model, it is especially important to have a

clear relationship between the formal definition of the programming language and that of the debugger.

In this paper, we begin to explore these issues by giving formal semantic definitions, of a simple strict functional programming language, and of a debugger for this language. The debugger allows the programmer to focus the scope of attention on a specific subexpression within the program, thereby circumventing in a controlled fashion the information hiding implied by λ-bound identifiers. We use a "transition-style" structured operational semantics to define the programming language and debugging constructs. The use of a transition-style semantics, rather than a "natural semantics" style, allows us to define in an explicit and intuitive fashion the notion of an evaluation step, which seems essential for describing the interaction between the debugger and the program being debugged. The transition-style semantics also lends itself well to describing the interaction between the programmer and the debugger. However, the construction of a transition-style semantics for a functional programming language was not accomplished without some difficulties, centering primarily around the problem of giving a transition-oriented definition of substitution while making sure that alpha-convertible terms remained behaviorally equivalent. We were unable to find any other examples in the literature of a transition-style semantics for a functional programming language. In fact, it seems that people have avoided transition-style semantics because of the kinds of problems we faced [5,6], and we suspect that ours is the first such definition.

The usual notion of semantic equivalence in a transition-style operational semantics is *bisimulation* [12]. In writing the semantics for our programming language, we have been careful to make sure that bisimulation yields a "minimally reasonable" notion of program equivalence. In particular, we show that α-convertible terms are bisimilar, and that bisimulation is a congruence with respect to the programming language constructs. In order to establish a clear relationship between the debugger and programming language, we define the debugger as an additional level of syntactic and semantic rules on top of those for the programming language. This extension is shown to be conservative, in the sense that the additional rules do not permit additional transitions to be inferred for programs in the underlying language. Furthermore, the stratified form of the definition means that the debugger must extract information from the program being debugged by "synchronizing" on the labels of the transitions executed by the program, rather than by directly inspecting the program syntax. We expect that a debugger defined in this way will lend itself to implementation through source code instrumentation.

The rest of this paper is organized as follows. In the remainder of this introductory section, we describe some related work and give some necessary preliminary definitions. In Section 2, we define the syntax for a simple strict functional programming language and present semantic rules that describe the evaluation steps for expressions in the language. We prove some "healthiness properties" for the language, to provide confidence that the semantics is reasonable. In Section 3, we describe the syntactic and semantic rules for

our debugging constructs. We then establish our main formal results about the debugger, namely that the debugging rules conservatively extend the programming language, that a program has the same behavior when it is being debugged as when it is not being debugged, and that "focusing" on a subexpression preserves the meaning of a program being debugged. A full version of this paper with complete proofs is available as a technical report [3].

1.1 Related work

Although there is a large literature on formal definitions of programming languages, comparatively little work has been done in applying formal techniques to designing debuggers. Shapiro introduced the first attempt to lay a theoretical framework for debugging in Prolog [14]. Shapiro's Algorithmic Debugger uses top-down analysis along with information from the programmer to automatically determine the section of code that contains a bug.

Kishon, Hudak and Consel introduced a semantic framework for describing and generating *program execution monitors* [10]. Monitors are tools such as debuggers, profilers, and tracers that can view the execution of a program. Kishon *et al.* presented a *monitoring semantics* as an extension to the continuation-passing denotational semantics for a language. Partial evaluation was used to generated the debugger from the specifications.

DaSilva described a method for specifying and proving correct compilers and debuggers based on structured operational semantics [6]. Because of the emphasis of his work was in proving correctness, he chose to use relational semantics and define an evaluation step as a secondary notion, rather than using transitional semantics and have an evaluation step be explicit.

Our work differs from that of Kishon *et al.* and from DaSilva in emphasis and approach. Our work focuses on designing novel debugging environments rather than generating or proving correct traditional debugging environments. We also try a different approach: whereas Kishon *et al.* used continuation passing style denotational semantics as their underlying formalism and DaSilva used relational operational semantics, we use transitional operational semantics. For a more detailed analysis of the relative advantages of each approach see [4].

Other researchers have treated bisimulation as a program equivalence for functional languages. Abramsky introduced applicative bisimulation as the notion for operational equality for the lazy λ-calculus [1]. Howe used bisimulation as his notion of equivalence for a class of lazy computation systems and demonstrated an elegant proof method for showing that bisimulation is a congruence [9]. Gordon used bisimulation for program equivalence in his work where he gives a semantics to I/O mechanisms for functional programming languages [7].

1.2 Preliminaries

In this paper, we use standard notions for term deduction systems (TDS) and their corresponding labeled transition systems (LTS). For complete formal

definitions see [8]. A *signature* consists of a set of function symbols along with a rank function that gives the arity for each function symbol. The set of terms defined by a signature Σ, over a set W of variables, is denoted $T(\Sigma, W)$. The set $T(\Sigma, \emptyset)$ is abbreviated $T(\Sigma)$ and elements of the set are called *ground* terms. A *term deduction system (TDS)* is a triple (Σ, A, R) with Σ a signature, A a set of labels, and R a set of rules of the form:

$$\frac{\{x_i \xrightarrow{\alpha_i} x_i' \mid i \in I\}}{x \xrightarrow{\alpha} x'}$$

where I is a finite index set, the x's are terms in $T(\Sigma, V)$ and the α's are labels. For $P = (\Sigma, A, R)$ a TDS, a *proof* from P of a transition ψ is a finite, upwardly branching tree whose nodes are labeled by transitions $x \xrightarrow{\alpha} x'$ such that: (1) the root is labeled with ψ, and (2) if χ is the label of a node q and $\{\chi_i | i \in I\}$ is the set of labels of the nodes directly above q, then there is a rule

$$\frac{\{\phi_i \mid i \in I\}}{\phi}$$

in R and a substitution $\sigma : V \to T(\Sigma, V)$ such that $\chi = \sigma(\phi)$ and $\chi_i = \sigma(\phi_i)$ for $i \in I$.

A *labeled transition system (LTS)* is a structure $(S, A, \to)$ where S is a set of *states*, A is set of *actions*, and $\to \subseteq S \times A \times S$ is a *transition relation.* For $s, t \in S$, we use $s \to^* t$ to mean there exists $s_i \in S$ $(0 \leq i \leq n)$ such that $s_0 \to s_1$, $s_1 \to s_2$, $s_2 \to s_3$, ..., $s_{n-1} \to s_n$, where $s_0 = s$ and $s_n = t$.

Let $\mathcal{A} = (S, A, \to)$ be a labeled transition system, then a relation $R \subseteq S \times S$ is a *(strong) bisimulation* if it satisfies:

$(s\ R\ t$ and $s \xrightarrow{\alpha} s')$ implies $(\exists t' \in S, t \xrightarrow{\alpha} t'$ and $s'\ R\ t')$; and

$(s\ R\ t$ and $t \xrightarrow{\alpha} t')$ implies $(\exists s' \in S, s \xrightarrow{\alpha} s'$ and $s'\ R\ t')$.

Two states $s, t \in S$ are *bisimilar* in $\mathcal{A}$ (denoted $\mathcal{A} : s \sim t$) if there exists a bisimulation relating them. For $P = (\Sigma, A, R)$ a TDS, the *transition system* $TS(P)$ *specified by* P is given by $TS(P) = (T(\Sigma), A, \to_P)$, where (x, α, x') is in $\to_P$ if and only if there exists a proof from P of $x \xrightarrow{\alpha} x'$.

2 Programming language

Our programming language is a simple strict functional language with a non-strict conditional expression. The syntax of our language is:

$k \in$ Constants $\qquad a \in$ Identifiers

$e \in$ Expressions $::= k \mid a \mid (\mathbf{fn}\ a => e) \mid$ if e_1 then e_2 else $e_3 \mid e_1\ e_2$

We regard the terms of the programming language as built up from primitive expressions using syntactic constructor functions. For example, the expression $(\mathbf{fn}\ a => 0)$ is built up by applying a binary constructor $(\mathbf{fn}\ - => -)$ to two arguments, the first of which is an identifier a, and the second of which is a constant 0. A *value* is an expression that is either a constant or an expression of the form $(\mathbf{fn}\ a => e)$.

We now give an operational semantics for our programming language, in the form of a term deduction system. In presenting this semantics, we use the following naming conventions: x, y, z, w are variables that range over terms; a, b denote identifiers; k denotes a constant; e denotes an expression; v denotes a value; and α denotes an arrow label. There are three groups of transitions in our language definition. *Typing transitions* serve to classify fully evaluated terms, *substitution transitions* perform syntactic substitution, and *evaluation transitions* are the actual evaluation steps.

The rules for typing transitions are given at the top of Figure 1. Each typing transition is labeled either by an identifier a, a constant k, the special symbol v, or the label $\Box e$, where $\Box$ is a special symbol and e is an expression.

Most of the typing rules in Figure 1 (except (tp4)) are actually rule schemata, which define a possibly infinite collection of actual rules. For example, (tp1) is a rule schema that defines a separate rule for each constant k. The rules defined by schema (tp5) are obtained by instantiating a to a particular identifier and e to a particular expression.

The intuition behind the typing rules is as follows: Each identifier and constant can perform a transition to announce its identity (rules tp1 and tp2). This has the effect of making identifiers and constants bisimilar if and only if they are identical. Constants and function definitions are fully evaluated and can perform a "v" transition (means value) to announce this fact (rules tp3 and tp4). Function definitions can do $\Box e$ transitions (tp5). Intuitively, the transition $x \xrightarrow{\Box e} x'$ means "x when applied to argument e becomes x'". As a result of (tp5), two function definitions will end up being bisimilar if and only if they give bisimilar results when applied to argument expressions.

Substitution rules (see middle of figure 1) have labels of the form $[e/a]$, where e is an expression and a is an identifier. The transition $x \xrightarrow{[e/a]} x'$ should be read "x with e substituted for a becomes x'." With this reading, the intuitive interpretation of the rules is straightforward, except perhaps for (sub5). The rule (sub5) performs a change of the bound identifier to make sure that free occurrences of b in e are not captured by performing the substitution. Rule (sub3) is a rule schema that defines a transition $b \xrightarrow{[e/a]} b$ for each expression e and each pair of distinct identifiers a and b. The "premise" $b \neq a$ in rule (sub5) is interpreted similarly.

Evaluation transitions (see bottom of figure 1) are unlabeled; or more precisely, they have a special null label that we do not bother to write. Rule schema (ap3) defines a separate rule for each value v and each expression e. The evaluation rules define a strict left-to-right order of evaluation for our language.

Notice that the semantics we have given is almost in *tyft* format [8]. One characteristic feature of *tyft* format is that the left-hand side of the conclusion of each rule is restricted to contain just a single function symbol. The only place we have violated this restriction in our rules is in (ap3), where the left-hand side of the conclusion contains two function symbols (application, and either a constant or a function definition). Expressing the rules in

Typing rules:

$$k \xrightarrow{k} k \quad (tp1) \qquad a \xrightarrow{a} a \quad (tp2) \qquad k \xrightarrow{\mathrm{v}} k \quad (tp3)$$

$$(\mathbf{fn}\ x \Rightarrow y) \xrightarrow{\mathrm{v}} (\mathbf{fn}\ x \Rightarrow y) \quad (tp4) \qquad \frac{x \xrightarrow{a} x' \quad y \xrightarrow{[e/a]} y'}{(\mathbf{fn}\ x \Rightarrow y) \xrightarrow{\Box e} y'} \quad (tp5)$$

Substitution rules:

$$k \xrightarrow{[e/a]} k \quad (sub1) \qquad a \xrightarrow{[e/a]} e \quad (sub2)$$

$$\frac{b \neq a}{b \xrightarrow{[e/a]} b} \quad (sub3) \qquad \frac{x \xrightarrow{[e/a]} x' \quad y \xrightarrow{[e/a]} y'}{x\ y \xrightarrow{[e/a]} x'\ y'} \quad (sub4)$$

$$\frac{x \xrightarrow{b} x' \quad y \xrightarrow{[b'/b]} y' \quad y' \xrightarrow{[e/a]} y'' \quad b \neq a}{(\mathbf{fn}\ x \Rightarrow y) \xrightarrow{[e/a]} (\mathbf{fn}\ b' \Rightarrow y'')} \ (sub5)$$

(*where b' is the first identifier name that does not occur free in* $(\mathbf{fn}\ x \Rightarrow y)$ *or* e)

$$\frac{x \xrightarrow{a} x'}{(\mathbf{fn}\ x \Rightarrow y) \xrightarrow{[e/a]} (\mathbf{fn}\ x \Rightarrow y)} \quad (sub6)$$

$$\frac{x \xrightarrow{[e/a]} x' \quad y \xrightarrow{[e/a]} y' \quad z \xrightarrow{[e/a]} z'}{\text{if } x \text{ then } y \text{ else } z \xrightarrow{[e/a]} \text{if } x' \text{ then } y' \text{ else } z'} \ (sub7)$$

Evaluation rules:

$$\frac{x \longrightarrow x'}{x\ y \longrightarrow x'\ y} \quad (ap1) \qquad \frac{x \xrightarrow{\mathrm{v}} x' \quad y \longrightarrow y'}{x\ y \longrightarrow x\ y'} \quad (ap2)$$

$$\frac{x \xrightarrow{\Box v} x'}{x\ v \longrightarrow x'} \quad (ap3) \qquad \frac{x \longrightarrow x'}{\text{if } x \text{ then } y \text{ else } z \longrightarrow \text{if } x' \text{ then } y \text{ else } z} \quad (if1)$$

$$\frac{x \xrightarrow{k} x' \quad k \neq 0}{\text{if } x \text{ then } y \text{ else } z \longrightarrow y} \quad (if2) \qquad \frac{x \xrightarrow{0} x'}{\text{if } x \text{ then } y \text{ else } z \longrightarrow z} \quad (if3)$$

Fig. 1. Operational semantics for programming language

this restricted fashion has the technical effect of simplifying structural induction proofs using the rules. In particular, a semantics expressed entirely in *tyft* format automatically has the property that bisimilarity is a congruence. We were unsuccessful at finding a completely *tyft*-format semantics for our language and we suspect that it is not possible to do so.

2.1 Properties of the Programming Language

The following proposition, which states that substitution transitions exactly correspond to syntactic substitution, is proved by structural induction on x:

Proposition 2.1 *For all expressions x, x', e and all identifiers a:*

$$x \xrightarrow{[e/a]} x' \iff x' = x[e/a].$$ □

We say that an expression x is *fully evaluated* if and only if no evaluation (unlabeled) transition $x \longrightarrow y$ is provable. Observe that all of the typing rules have a constant, identifier or a function definition as the function symbol on the left-hand side of the conclusion, whereas the evaluation rules all have an application or a conditional as the function symbol on the left-hand side of the conclusion. Therefore, if a typing transition is provable for a term, then the term must either be an identifier, or have a constant or a function definition as its outermost function symbol, in which case no evaluation transition is provable for it. We have thus shown:

Proposition 2.2 *For all expressions x, if a typing transition $x \xrightarrow{\alpha} y$ is provable, then x is fully evaluated.* □

We can show, using structural induction, that the semantics we have defined is deterministic:

Proposition 2.3 *For all expressions x, at most one evaluation transition $x \longrightarrow y$ is provable.* □

We say that an expression x *evaluates to* an expression y, and we write $x{\downarrow}y$, if $x \rightarrow^* y$ and y is fully evaluated.

Corollary 2.4 *For all expressions x, there is at most one expression y such that $x{\downarrow}y$.* □

Our semantics is somewhat unusual in the sense that transition labels in many cases contain expressions of the programming language. In doing this, we run the risk that too much of the syntactic structure of a term might be exposed by the transition labels, making bisimilarity insufficiently abstract, and therefore an uninteresting, equivalence on expressions. Although we do not have a full characterization of bisimilarity for our language, the next result shows that at least the worst does not happen. The proof is accomplished by considering the relation that relates all terms that are identical up to the renaming of bound identifiers and showing that it is in fact a bisimulation relation.

Proposition 2.5 *If expressions x and y are identical up to renaming of bound identifiers, then they are bisimilar.* □

Finally, bisimilarity is compatible with the constructs of our language (*i.e.* is a congruence):

Proposition 2.6 *For all contexts $C[\ ]$ and all expressions $x, y \in T(\Sigma)$, if $x \sim y$ then $C[x] \sim C[y]$.* □

A full proof of this result is given in [3]. Our proof, done from "first principles," was not straightforward to find, essentially due to the failure of the rules (ap3) to be in *tyft* format and the need to find a bisimulation relation that is closed under substitution. One of the anonymous referees pointed out a technique, recently discovered by Howe [9], which elegantly handles the problem of obtaining bisimulations that are closed under substitution. We believe that Howe's technique can be used to factor out the technical portions of our proof having to do with the construction of the bisimulation relation, thereby simplifying the presentation. However, in comparing Howe's technique with our proof, it appears to us that the essentials ideas remain the same, whether or not Howe's method is used. In particular, we do not believe that Howe's technique leads to any simplification in the transition rules for the programming language. On the contrary, some of the common elements between Howe's approach and our transition rules suggest that the approach we have taken in formulating the transition rules is reasonable, and perhaps even essential in order for bisimilarity to be a congruence.

3 Debugging language

In this section we define a debugger which provides the novel capability of allowing the programmer to "focus" or shift the scope of attention in a syntax-directed fashion to a specific subexpression within the program, and to view the execution of the program from that vantage. In particular, we extend the syntax of the programming language with a new construct, a *focusing operator* ($\langle\!\{\ \}\!\rangle$) that allows the scope of attention to be focused on the evaluation of a particular subexpression. The focusing operator is applied to two argument expressions: an ordinary programming language expression (written in between the brackets), and a "debugging context" (written outside the brackets to the left). Debugging starts out with an empty debugging context and the entire program in focus between the brackets. The programmer then has a choice, either of observing execution steps, or of applying "focusing" operations to select a particular subexpression of the program to observe. Applying a focusing operation has the effect of moving the brackets onto a subexpression of the program, and of moving into the debugging context the portion of the program expression no longer within the scope of the brackets.

It is our intention that, during execution, only evaluation steps for the portion of the program in focus would be directly observable by the programmer. These steps should occur as described by the semantic definition of the programming language. Evaluation steps for the debugging context occur

silently or unobservably—"in the background," so to speak. The main goal of the semantic definition for the debugger is to describe the possible background transitions of the debugging context and the way in which the debugging context interacts with the portion of the program in focus, in such a way that the overall execution agrees with the programming language definition. The key property that we wish to ensure is that "focusing preserves meaning." By this we mean that the programmer can choose any subexpression as the focus of attention, and can even interleave focusing operations with evaluation steps, but the overall course of evaluation of the program remains as it would be if the program were not being debugged.

As stated in the introduction, our goal is to define the debugger "on top of" the programming language definition, as an additional level of syntactic and semantic rules on top of those for the programming language. This extension is shown to be conservative, in the sense that the additional debugging rules do not permit additional transitions to be inferred for programs in the underlying language. Furthermore, the stratified form of the definition means that the debugger must extract information from the program being debugged by "synchronizing" on the labels of the transitions executed by the program, rather than by directly inspecting the program syntax.

Formally, the syntax for our language is extended with the following debugging constructs:

$$c \in \text{ Coexps } ::= \{ \text{ - } \ e\} \mid \{e \ \text{ - } \} \mid \{ \text{ - } ?e_1, e_2\} \mid \{e_1? \text{ - } , e_2\} \mid \{e_1?e_2, \text{ - } \} \mid \{e/a\}$$

$$\kappa \in \text{ Contexts } ::= \epsilon \mid \kappa{:}\, c \qquad\qquad d \in \text{ DBStates } ::= \kappa\langle\!\langle x \rangle\!\rangle$$

A *debugging state* ($\kappa\langle\!\langle x \rangle\!\rangle$) consists of an expression x from the programming language together with a *debugging context* κ, which is a list of "coexpressions." A *coexpression*, in turn, is either an ordinary programming language expression that has a designated missing subterm (*e.g.* $\{ \text{ - } \ e\}$ or $\{e \ \text{ - } \}$), or else is a *substitution coexpression* ($\{e/a\}$) indicating that we have moved within the scope of a substitution that has yet to be applied. The coexpressions corresponding to the conditional statement (if e_1 then e_2 else e_3) are abbreviated $\{ \text{ - } ?e_2, e_3\}$, $\{e_1? \text{ - } , e_3\}$, and $\{e_1?e_2, \text{ - } \}$.

Any term x in the programming language can be packaged together with an empty debugging context ϵ to form a debugging state $\epsilon\langle\!\langle x \rangle\!\rangle$. The transition rules for the debugger are defined in such a way that the term x and the term $\epsilon\langle\!\langle x \rangle\!\rangle$ evaluate the same way.

The debugger allows the focus of attention to be shifted to a particular subexpression through *focusing operations*. The various focusing operations are defined in Figure 2. These operations can be viewed as explicit actions taken by the programmer to modify the debugging state in order to select the focus of attention. For example, "focusing left" ($\overset{l}{\Downarrow}$) on an application focuses the scope of attention on the operator and places an application coexpression containing the operand into the debugging context.

Transitions labeled by $\overset{\text{fn}}{\Downarrow}$ are unusual in that not only do they require that

$$\kappa\langle\!|x\ y|\!\rangle \overset{l}{\Downarrow} \kappa{:}\{\ -\ \ y\}\langle\!|x|\!\rangle$$

$$\kappa\langle\!|x\ y|\!\rangle \overset{r}{\Downarrow} \kappa{:}\{x\ \ -\ \}\langle\!|y|\!\rangle$$

$$\kappa\langle\!|\text{if } x \text{ then } y \text{ else } z|\!\rangle \overset{\text{if}}{\Downarrow} \kappa{:}\{\ -\ ?y, z\}\langle\!|x|\!\rangle$$

$$\kappa\langle\!|\text{if } x \text{ then } y \text{ else } z|\!\rangle \overset{\text{then}}{\Downarrow} \kappa{:}\{x?\ -\ , z\}\langle\!|y|\!\rangle$$

$$\kappa\langle\!|\text{if } x \text{ then } y \text{ else } z|\!\rangle \overset{\text{else}}{\Downarrow} \kappa{:}\{x?y,\ -\ \}\langle\!|z|\!\rangle$$

$$\kappa{:}\{\ -\ \ y\}\langle\!|(\mathbf{fn}\ a => x)|\!\rangle \overset{\text{fn}}{\Downarrow} \kappa{:}\{y/a'\}\langle\!|x[a'/a]|\!\rangle$$

where a' does not occur in $\kappa{:}\{\ -\ \ y\}\langle\!|(\mathbf{fn}\ a => x)|\!\rangle$

Fig. 2. Focusing rules

the expression in the focus of attention be a function definition, but also the rightmost coexpression in the debugging context must correspond to an application with an operand. This situation reflects the fact that function definitions have first-class status in our programming language. That is, a function definition can be used either as an operator in an application or simply as a fully evaluated data value. If a function definition is used as operator in an application, then focusing inside the function definition corresponds to binding the operand to the identifier specified by the function definition, and then focusing on the function body. On the other hand, if the function definition is used as a fully evaluated data value then focusing inside it makes no more sense than focusing inside any other constant.

The evaluation of debugging states is defined by several different groups of transition rules. Here we present only some of the rules; the full set of rules is available in the appendix. Although there are a substantial number of rules, we wish to point out that they are mostly forced by our choice of the focusing rules and our desire that "focusing preserves meaning." In a sense, the transition rules for the debugger show that our choice of debugging state is adequate to satisfy our requirements. The challenge in writing the debugger definition was to identify the proper notion of debugging state.

For clarity in the presentation of the rules, we use a double arrow ($\Longrightarrow$) to distinguish transitions inferred using the debugging rules from those inferred using the programming language rules. In the debugger, as in the underlying programming language, unlabeled transitions once again correspond to evaluation steps, transitions labeled with "v", k, $\Box e$, or a correspond to typing steps and transitions labeled with $[e/a]$ correspond to substitution steps. Transitions labeled with "!" are called *trigger* transitions. These transitions of the debugging context serve to trigger, or control, the evaluation of the expression in focus. The reason the debugging context needs to exercise this control is because we want to make sure that the same evaluation order ap-

plies to a program when it is being debugged as when it is not being debugged. The transitions labeled with "$*$" are used for special handling of conditional expressions that appear within a debugging context. If the focus of attention is moved inside one of the arms of a conditional expression before the condition has been fully evaluated, then it is not known whether or not the chosen arm is the one that will actually be executed. If the chosen arm is not the one that will actually be executed, then at the point where the condition becomes fully evaluated, a "$*$" transition executed by the debugging context will serve to replace the useless debugging state containing the wrong branch of the conditional by a new debugging state containing the correct branch.

The following rules define evaluation steps for debugging states. Evaluation can occur within the debugging context (db1) or, if triggered by the debugging context, within the expression in focus (db2). Labeled transitions for the expression in focus "synchronize" with complementary transitions of the debugging context, to ensure that the overall evaluation is consistent with the programming language definition (db3). Finally, as already mentioned, if evaluation within the debugging context determines that the expression in focus is in the wrong arm of a conditional, then the "wrong" debugging context is replaced by the correct one (db9).

$$\frac{\kappa \Longrightarrow \kappa'}{\kappa\langle x\rangle \Longrightarrow \kappa'\langle x\rangle} \quad (db1) \qquad \frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \longrightarrow x'}{\kappa\langle x\rangle \Longrightarrow \kappa\langle x'\rangle} \quad (db2)$$

$$\frac{\kappa \overset{[e/a]}{\Longrightarrow} \kappa' \quad x \overset{[e/a]}{\longrightarrow} x'}{\kappa\langle x\rangle \Longrightarrow \kappa'\langle x'\rangle} \quad (db3) \qquad \frac{\kappa \overset{*}{\Longrightarrow} d}{\kappa\langle x\rangle \Longrightarrow d} \quad (db9)$$

The following rules define the evaluation steps for debugging contexts. Rule (ke1) says that any evaluation step that can be executed by a debugging context can still be executed even if an additional coexpression is appended. Rule (ke2) states that coexpressions can perform evaluation steps in a fashion consistent with the programming language definition, as long as these steps are permitted by the debugging context to their left. Finally, rule (ke8) states that substitution transitions are hidden by substitution coexpressions for the same bound variable. This rule corresponds to rule (sub6) for the programming language.

$$\frac{\kappa \Longrightarrow \kappa'}{\kappa\colon x \Longrightarrow \kappa'\colon x} \quad (ke1) \qquad \frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \longrightarrow x'}{\kappa\colon\{x\ \text{-}\} \Longrightarrow \kappa\colon\{x'\ \text{-}\}} \quad (ke2)$$

$$\frac{\kappa \overset{[e/a]}{\Longrightarrow} \kappa' \quad x \overset{[e/a]}{\longrightarrow} x'}{\kappa\colon\{x/a\} \Longrightarrow \kappa'\colon\{x'/a\}} \quad (ke8)$$

The following rules specify how conditionals are evaluated within a debugging context, in the event that the expression in focus is in the "wrong" arm of the conditional. If the test expression evaluates to 0, but the focus is on the "then" branch, then the current debugging context is abandoned and a new debugging context containing the "else" branch is installed (br2). The case in which the test expression evaluates to a nonzero value, but the focus

is one the "else" branch is similar (br3). Rule (br1) has the effect of deleting any coexpressions in the debugging context that pertain to the "wrong" arm of the conditional.

$$\frac{\kappa \stackrel{*}{\Longrightarrow} d}{\kappa{:}\, x \stackrel{*}{\Longrightarrow} d}\ (br1) \qquad \frac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad x \stackrel{0}{\longrightarrow} x'}{\kappa{:}\,\{x?\ \text{-}\ , z\} \stackrel{*}{\Longrightarrow} \kappa\langle z\rangle}\ (br2)$$

$$\frac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad x \stackrel{k}{\longrightarrow} x' \quad k \neq 0}{\kappa{:}\,\{x?y,\ \text{-}\ \} \stackrel{*}{\Longrightarrow} \kappa\langle y\rangle}\ (br3)$$

The following rules describe the generation and propagation of control information within a debugging context. These rules ensure that a program evaluates in the same order when it is being debugged as when it is not being debugged.

$$\epsilon \stackrel{!}{\Longrightarrow} \epsilon\ \ (tr0) \qquad \frac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad x \stackrel{\mathrm{v}}{\longrightarrow} x'}{\kappa{:}\,\{x\ \text{-}\ \} \stackrel{!}{\Longrightarrow} \kappa{:}\,\{x\ \text{-}\ \}}\ (tr1)$$

$$\frac{\kappa \stackrel{!}{\Longrightarrow} \kappa'}{\kappa{:}\,\{\ \text{-}\ \ y\} \stackrel{!}{\Longrightarrow} \kappa{:}\,\{\ \text{-}\ \ y\}}\ (tr2)$$

The last set of rules specifies how substitutions are applied to debugging contexts. Application of substitutions is controlled by trigger transitions from the debugging context to the left (sb1). Once triggered, substitutions propagate to the right, applying themselves to any coexpressions they encounter (sb2).

$$\frac{\kappa \stackrel{!}{\Longrightarrow} \kappa'}{\kappa{:}\,\{v/a\} \stackrel{[v/a]}{\Longrightarrow} \kappa}\ (sb1) \qquad \frac{\kappa \stackrel{[e/a]}{\Longrightarrow} \kappa' \quad x \stackrel{[e/a]}{\longrightarrow} x'}{\kappa{:}\,\{\ \text{-}\ \ x\} \stackrel{[e/a]}{\Longrightarrow} \kappa'{:}\,\{\ \text{-}\ \ x'\}}\ (sb2)$$

As a simple example of the use of the debugging rules, consider the case where the constant function (**fn** $a \Rightarrow 1$) is applied to the argument 2. From the semantic definition of the programming language we can infer the following transition for the expression:

$$\frac{\dfrac{a \stackrel{a}{\longrightarrow} a \quad 1 \stackrel{[2/a]}{\longrightarrow} 1}{(\mathbf{fn}\ a \Rightarrow 1) \stackrel{\Box 2}{\longrightarrow} 1}\ (tp5)}{(\mathbf{fn}\ a \Rightarrow 1)\ 2 \longrightarrow 1}\ (ap3)$$

Now, suppose that we wish to debug this expression with the scope of attention focused on the body of the function. We start from the debugging expression with the empty debugging context and focus left on the function definition. This results in the operand 2 moving into the debugging context as the coexpression $\{\ \text{-}\ \ 2\}$. We can then focus our attention on the function body, which causes the argument 2 to combine with the λ-bound identifier a

to yield the substitution $\{2/a\}$ in the debugging context:

$$\epsilon\langle\!|(\mathbf{fn}\ a => 1)\ 2|\!\rangle \overset{l}{\Downarrow} \epsilon{:}\{\ -\ \ 2\}\langle\!|(\mathbf{fn}\ a => 1)|\!\rangle \overset{\mathrm{fn}}{\Downarrow} \epsilon{:}\{2/a\}\langle\!|1|\!\rangle.$$

In the corresponding transition of the debugging state, we see the substitution get triggered and propagate through the debugging expression:

$$\dfrac{\dfrac{\dfrac{}{\epsilon \overset{!}{\Longrightarrow} \epsilon}\ (tr0)}{\epsilon{:}\{2/a\} \overset{[2/a]}{\Longrightarrow} \epsilon}\ (sb1) \qquad \dfrac{}{1 \overset{[2/a]}{\longrightarrow} 1}\ (sub1)}{\epsilon{:}\{2/a\}\langle\!|1|\!\rangle \Longrightarrow \epsilon\langle\!|1|\!\rangle}\ (db3)$$

3.1 Properties of the Debugger

In this section, we establish some results that show the definitions we have given for the debugger are sensible. The first result states that the debugging rules conservatively extend those of the programming language, in the sense that no transitions can be inferred for a program using the debugging rules, that cannot already be inferred for that program using the programming language rules alone. The result is a simple consequence of the fact that the left-hand side of the conclusion of each debugging rule contains a function symbol that is not a function symbol of the programming language.

Proposition 3.1 *For all programming language expressions x, a transition $x \xrightarrow{\alpha} y$ is provable using the programming language rules and the debugger rules if and only if it is provable using the programming language rules alone.*□

The second result states that bisimilar expressions placed in the same debugging context yield bisimilar debugging states. Because of the stratified approach to the debugger definition, this result can be shown by a straightforward case analysis on the possible debugging transitions.

Proposition 3.2 *For all programming language expressions x, x', if $x \sim x'$ then for all debugging contexts κ, $\kappa\langle\!|x|\!\rangle \sim \kappa\langle\!|x'|\!\rangle$.* □

If X is a subset of the set of transitions of a transition system, then define two states q and r to be *bisimilar excluding X transitions*, if q and r are bisimilar in the transition system obtained by deleting all transitions in X from the original transition system. The next result states that a program evaluates the same way in an empty debugging context as it does when it is not being debugged. A particular consequence of this result is that, if a program x evaluates to a constant k, then debugging state $\epsilon\langle\!|x|\!\rangle$ evaluates to $\epsilon\langle\!|k|\!\rangle$.

Proposition 3.3 *For all programming language expressions x, x is bisimilar to $\epsilon\langle\!|x|\!\rangle$, excluding $[e/a]$-labeled transitions.* □

Our main result is the following, which states that "focusing preserves meaning," in the sense that shifting the focus of attention in a debugging

state results in a new debugging state that is bisimilar (excluding $[e/a]$-labeled transitions) to the original one. The result is proved by a detailed case analysis of the possible focusing operations and transitions.

Theorem 3.4 *For all debugging states d and d', if $d \overset{\delta}{\Downarrow} d'$, then d is bisimilar to d', excluding $[e/a]$-labeled transitions.* □

4 Conclusion

In this paper we presented a transition-style operational semantics for a simple functional language and an associated debugger for that language. The debugger provides the novel capability of allowing the programmer to focus the scope of attention in a syntax directed fashion. Our main formal result was that "focusing preserves meaning" that is a program exhibits bisimilar behavior regardless of the subexpression in focus. To achieve this result, we faced some interesting issues, such as how to build a semantic definition of a debugger on top of a programming language definition, how to give a transition-style structured operational semantics for a functional programming language, and what sort of formal relationships ought to hold between the definition of a debugger and that of the underlying programming language. For the next step in this research, we are working on using our semantic definition as the foundation for implementing a debugger for a subset of SML.

References

[1] Samson Abramsky. The lazy lambda calculus. In David Turner, editor, *Research Topics in Functional Programming*, The UT Year of Programming Series, pages 65–116. Addison-Wesley Publishing Company, 1990.

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, Cambridge, 1992.

[3] Karen L. Bernstein and Eugene W. Stark. Operational semantics of a focusing debugger (full version). Technical report, State University of New York at Stony Brook, Computer Science Department, 1994. FTP://ftp.cs.sunysb.edu/pub/TechReports/stark/focus.ps.Z.

[4] Karen L. Bernstein and Eugene W. Stark. On formally defining debuggers. In *2nd International Workshop on Automated and Algorithmic Debugging*, St. Malo, France, May 1995.

[5] Dave Berry. *Generating Program Animators from Programming Language Semantics.* PhD thesis, University of Edinburgh, Edinburgh, Scotland, June 1991. LFCS, Department of Computer Science.

[6] 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.

[7] Andrew Gordon. An operational semantics for I/O in a lazy functional language. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture*, pages 136–145. Association for Computing Machinery, ACM Press, 1993.

[8] Jan Friso Groote and Frits Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100:202–260, 1992.

[9] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 198–203. IEEE, IEEE Computer Society Press, 1989.

[10] 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.

[11] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[12] D. M. R. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science.* Springer-Verlag, 1981.

[13] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.

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

[15] 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.

A Debugging Rules

$$\frac{\kappa \Longrightarrow \kappa'}{\kappa\langle x\rangle \Longrightarrow \kappa'\langle x\rangle} \quad (db1) \qquad \frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \longrightarrow x'}{\kappa\langle x\rangle \Longrightarrow \kappa\langle x'\rangle} \quad (db2)$$

$$\frac{\kappa \overset{[e/a]}{\Longrightarrow} \kappa' \quad x \overset{[e/a]}{\longrightarrow} x'}{\kappa\langle x\rangle \Longrightarrow \kappa'\langle x'\rangle} \quad (db3) \qquad \frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \overset{\Box v}{\longrightarrow} x'}{\kappa\!:\!\{\text{ - } v\}\langle x\rangle \Longrightarrow \kappa\langle x'\rangle} \quad (db4)$$

$$\frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \overset{v}{\longrightarrow} x' \quad y \longrightarrow y'}{\kappa\!:\!\{\text{ - } y\}\langle x\rangle \Longrightarrow \kappa\!:\!\{\text{ - } y'\}\langle x\rangle} \quad (db5)$$

$$\frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \overset{\Box v}{\longrightarrow} x'}{\kappa\!:\!\{x \text{ - }\}\langle v\rangle \Longrightarrow \kappa\langle x'\rangle} \quad (db6) \qquad \frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \overset{k}{\longrightarrow} x' \quad k \neq 0}{\kappa\!:\!\{\text{ - } ?y, z\}\langle x\rangle \Longrightarrow \kappa\langle y\rangle} \quad (db7)$$

$$\frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \overset{0}{\longrightarrow} x'}{\kappa\!:\!\{\text{ - } ?y, z\}\langle x\rangle \Longrightarrow \kappa\langle z\rangle} \quad (db8) \qquad \frac{\kappa \overset{*}{\Longrightarrow} d}{\kappa\langle x\rangle \Longrightarrow d} \quad (db9)$$

Fig. A.1. Evaluation rules for debugging states

$$\frac{\kappa \Longrightarrow \kappa'}{\kappa\!:\!x \Longrightarrow \kappa'\!:\!x} \quad (ke1) \qquad \frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \longrightarrow x'}{\kappa\!:\!\{x \text{ - }\} \Longrightarrow \kappa\!:\!\{x' \text{ - }\}} \quad (ke2)$$

$$\frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \longrightarrow x'}{\kappa\!:\!\{x/a\} \Longrightarrow \kappa\!:\!\{x'/a\}} \quad (ke3) \qquad \frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \longrightarrow x'}{\kappa\!:\!\{x? \text{ - }, z\} \Longrightarrow \kappa\!:\!\{x'? \text{ - }, z\}} \quad (ke4)$$

$$\frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \overset{k}{\longrightarrow} x' \quad k \neq 0}{\kappa\!:\!\{x? \text{ - }, z\} \Longrightarrow \kappa} \quad (ke5) \qquad \frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \longrightarrow x'}{\kappa\!:\!\{x?y, \text{ - }\} \Longrightarrow \kappa\!:\!\{x'?y, \text{ - }\}} \quad (ke6)$$

$$\frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \overset{0}{\longrightarrow} x'}{\kappa\!:\!\{x?y, \text{ - }\} \Longrightarrow \kappa} \quad (ke7) \qquad \frac{\kappa \overset{[e/a]}{\Longrightarrow} \kappa' \quad x \overset{[e/a]}{\longrightarrow} x'}{\kappa\!:\!\{x/a\} \Longrightarrow \kappa'\!:\!\{x'/a\}} \quad (ke8)$$

Fig. A.2. Evaluation rules for debugging contexts

$$\frac{x \xrightarrow{k} x'}{\epsilon\langle\!|x|\!\rangle \overset{k}{\Longrightarrow} \epsilon\langle\!|x'|\!\rangle}\ (dt1) \qquad \frac{x \xrightarrow{a} x'}{\epsilon\langle\!|x|\!\rangle \overset{a}{\Longrightarrow} \epsilon\langle\!|x'|\!\rangle}\ (dt2)$$

$$\frac{x \xrightarrow{\mathrm{v}} x'}{\epsilon\langle\!|x|\!\rangle \overset{\mathrm{v}}{\Longrightarrow} \epsilon\langle\!|x'|\!\rangle}\ (dt3) \qquad \frac{x \xrightarrow{\Box e} x'}{\epsilon\langle\!|x|\!\rangle \overset{\Box e}{\Longrightarrow} \epsilon\langle\!|x'|\!\rangle}\ (dt4)$$

Fig. A.3. Typing rules for debugging states

$$\frac{\kappa \overset{*}{\Longrightarrow} d}{\kappa\colon x \overset{*}{\Longrightarrow} d}\ (br1) \qquad \frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \xrightarrow{0} x'}{\kappa\colon\{x?\ \text{-}\ , z\} \overset{*}{\Longrightarrow} \kappa\langle\!|z|\!\rangle}\ (br2)$$

$$\frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \xrightarrow{k} x' \quad k \neq 0}{\kappa\colon\{x?y,\ \text{-}\} \overset{*}{\Longrightarrow} \kappa\langle\!|y|\!\rangle}\ (br3)$$

Fig. A.4. Conditional branching rules for debugging contexts

$$\epsilon \overset{!}{\Longrightarrow} \epsilon\ \ (tr0) \qquad \frac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \xrightarrow{\mathrm{v}} x'}{\kappa\colon\{x\ \text{-}\} \overset{!}{\Longrightarrow} \kappa\colon\{x\ \text{-}\}}\ (tr1)$$

$$\frac{\kappa \overset{!}{\Longrightarrow} \kappa'}{\kappa\colon\{\ \text{-}\ y\} \overset{!}{\Longrightarrow} \kappa\colon\{\ \text{-}\ y\}}\ (tr2) \qquad \frac{\kappa \overset{!}{\Longrightarrow} \kappa'}{\kappa\colon\{\ \text{-}\ ?y, z\} \overset{!}{\Longrightarrow} \kappa\colon\{\ \text{-}\ ?y, z\}}\ (tr3)$$

Fig. A.5. Trigger rules for debugging contexts

$$\frac{\kappa \overset{!}{\Longrightarrow} \kappa'}{\kappa{:}\{v/a\} \overset{[v/a]}{\Longrightarrow} \kappa} \quad (sb1) \qquad \frac{\kappa \overset{[e/a]}{\Longrightarrow} \kappa' \quad x \overset{[e/a]}{\longrightarrow} x'}{\kappa{:}\{\text{ - } x\} \overset{[e/a]}{\Longrightarrow} \kappa'{:}\{\text{ - } x'\}} \quad (sb2)$$

$$\frac{\kappa \overset{[e/a]}{\Longrightarrow} \kappa' \quad x \overset{[e/a]}{\longrightarrow} x'}{\kappa{:}\{x \text{ - }\} \overset{[e/a]}{\Longrightarrow} \kappa'{:}\{x' \text{ - }\}} \quad (sb3) \qquad \frac{\kappa \overset{[e/a]}{\Longrightarrow} \kappa' \quad y \overset{[e/a]}{\longrightarrow} y' \quad z \overset{[e/a]}{\longrightarrow} z'}{\kappa{:}\{\text{ - } ?y, z\} \overset{[e/a]}{\Longrightarrow} \kappa'{:}\{\text{ - } ?y', z'\}} \quad (sb4)$$

$$\frac{\kappa \overset{[e/a]}{\Longrightarrow} \kappa' \quad x \overset{[e/a]}{\longrightarrow} x' \quad z \overset{[e/a]}{\longrightarrow} z'}{\kappa{:}\{x? \text{ - }, z\} \overset{[e/a]}{\Longrightarrow} \kappa'{:}\{x'? \text{ - }, z'\}} \quad (sb5) \qquad \frac{\kappa \overset{[e/a]}{\Longrightarrow} \kappa' \quad x \overset{[e/a]}{\longrightarrow} x' \quad y \overset{[e/a]}{\longrightarrow} y'}{\kappa{:}\{x?y, \text{ - }\} \overset{[e/a]}{\Longrightarrow} \kappa'{:}\{x'?y', \text{ - }\}} \quad (sb6)$$

$$\frac{\kappa \overset{[e/a]}{\Longrightarrow} \kappa' \quad x \overset{[e/a]}{\longrightarrow} x' \quad a \neq b}{\kappa{:}\{x/b\} \overset{[e/a]}{\Longrightarrow} \kappa'{:}\{x'/b\}} \quad (sb7)$$

Fig. A.6. Substitution rules for debugging contexts