Operational Semantics of a Focusing Debugger (Full version)

Karen L. Bernstein, Eugene W. Stark

Department of Computer Science

State University of New York at Stony Brook

Stony Brook, NY 11794-4400 USA*

November 19, 1994

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.

Contact author: Karen L. Bernstein

Topics: Operational Semantics and Debuggers

*authors' E-mail addresses: `karen@cs.sunysb.edu`, `stark@cs.sunysb.edu`

1 Introduction

This paper explores two main ideas. First is the idea that 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, so that the set of concepts the programmer uses while debugging a program is essentially the same as the set of concepts the programmer had to use while writing the program. For example, a programmer should not be required to think in terms of a stack-based implementation while debugging a program in a language that is defined in terms of a rewriting semantics. 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 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 [Plo81], 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 [MTH90]. 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 [App92] 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 [Tol92] 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 doesn't 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? It seems reasonable to assume that the mental burden carried by the programmer will be reduced if the same conceptual model is used during the debugging of a program as is used when writing the code in the first place. Since the description of the conceptual model underlying a programming language is properly accomplished by a formal semantics for that language, it seems natural, then, that the conceptual model underlying a debugger should also be described by a formal semantics, and that the formal semantics for the debugger ought to be closely related to the formal semantics for the programming language itself. Further, the model used for the debugger should be at least as "high-level" as that used for the programming language, in the sense that debugging should not distinguish two programs which the programming language semantics regards as equivalent.

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.

The usual notion of semantic equivalence in a transition-style operational semantics is *bisimulation* [Par81]. 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 ensure that the debugger is at least as "high-level" as the 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 more readily to implementation through source code instrumentation.

There were some difficulties in using a transition-style semantics and it seems people have avoided transition-style semantics because of these kinds of problems [dS91, Ber91]. In order to define a semantics where we could prove some interesting properties, we tried to restrict the rules to be in *tyft* format [GV92]. We were unsuccessful at finding a completely *tyft*-format semantics and we suspect that it is not possible to do it. In particular, it was difficult to define syntactic substitution and still keep the semantics suitable abstract. As a result, the proof that bisimulation is a congruence was difficult. We were unable to find any other examples of a transition-style semantics for a functional programming language and we suspect that this is the first such definition.

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 and that "focusing" on a subexpression preserves the meaning of a program being debugged. The appendix contains the full versions of the proofs that were only outlined in the main sections of the paper.

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 [Sha83]. 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* [KHC91] [Kis92]. 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 [dS91]. 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.

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 [GV92]. 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

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

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 $s_0 \to s_1$, $s_1 \to s_2$, $s_2 \to s_3$, $\ldots$, $s_{n-1} \to s_n$, 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 $t \xrightarrow{\alpha} t')$ 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:

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

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

We regard the terms of the programming language as built up from primitive expressions using syntactic constructor functions. For example, the expression $(\textbf{fn}\ a \Rightarrow 0)$ is built up by applying a binary constructor (**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 (**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 in 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 rules in Figure 1 (except (tp4))

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

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

Figure 1: Typing rules

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" (means value) transition 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 figure 2) 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 rule (sub5). Rule (sub5) performs a change of name for the bound identifier in order to avoid capturing free occurrences of identifiers when performing the substitution.

Rule (sub3) is a rule schema which 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.

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

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

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

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

$$\frac{x \xrightarrow{a} x'}{(\mathbf{fn}\ x => y) \xrightarrow{[e/a]} (\mathbf{fn}\ x => y)} \quad (sub6) \qquad \frac{x \xrightarrow{[e/a]} x' \quad y \xrightarrow{[e/a]} y' \quad z \xrightarrow{[e/a]} z'}{\text{if } x \text{ then } y \text{ else } z \xrightarrow{[e/a]} \text{if } x' \text{ then } y' \text{ else } z'} \quad (sub7)$$

Figure 2: Substitution rules

Evaluation transitions (see figure 3) are unlabeled; that is to say, 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 right-to-left order of evaluation for our language.

Notice that the semantics we have given is almost in *tyft* format [GV92]. The only exceptions are the rules defined by rule schema (ap3), since in general the left-hand side of the conclusions of those rules will contain more than one function symbol. Expressing the semantics in this way is a help in proving theorems about it; for example, that bisimilarity is a congruence with respect to the programming language constructs. Another motivating idea behind the semantics is to make sure that transition labels do not expose too much syntactic information about the expression that is executing. In particular, we want α-convertible terms to be bisimilar. We were unsuccessful at finding a completely *tyft*-format semantics that has this property, and we suspect that it is not possible to do it.

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:

$$\frac{x \longrightarrow x' \quad y \stackrel{\mathrm{v}}{\longrightarrow} y'}{x\ y \longrightarrow x'\ y}\ (ap1) \qquad \frac{y \longrightarrow y'}{x\ y \longrightarrow x\ y'}\ (ap2) \qquad \frac{x \stackrel{\Box v}{\longrightarrow} x'}{x\ v \longrightarrow x'}\ (ap3)$$

$$\frac{x \longrightarrow x'}{\text{if } x \text{ then } y \text{ else } z \longrightarrow \text{if } x' \text{ then } y \text{ else } z}\ (if1)$$

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

Figure 3: Evaluation rules

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

$$x \stackrel{[e/a]}{\longrightarrow} x' \iff x' = x[e/a].$$

Proof Outline: The proof is by structural induction on x. See the appendix for the full version of the proof.

□

We say that an expression x is *fully evaluated* if and only if no evaluation (unlabeled) transition $x \longrightarrow y$ is provable.

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

Proof:

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. 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, the the term must have an outermost function symbol of a constant, an identifier, or a function definition and no evaluation transition is provable.

□

The semantics we have defined is deterministic:

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

Proof:

The proof by is structural induction on x. If the outermost function symbol is a constant, function definition, or an identifier then no evaluation transition is provable since no evaluation rules have any of these function symbols on the left-hand side of the conclusion. Therefore we only need to consider the cases where the outermost function symbol in x is either an application or a conditional expression.

If the outermost function symbol is an application of the form $(u\ v)$ then any provable transition will have a proof of one of the following forms:

$$\frac{\dfrac{\vdots}{u \longrightarrow u'} \quad v \xrightarrow{\mathrm{v}} v'}{u\ v \longrightarrow u'\ v}\ (\mathit{ap1}) \qquad \frac{\dfrac{\vdots}{v \longrightarrow v'}}{u\ v \longrightarrow u\ v'}\ (\mathit{ap2}) \qquad \frac{\dfrac{\vdots}{u \xrightarrow{\Box v} u'}}{u\ v \longrightarrow u'}\ (\mathit{ap3})$$

In the first proof an evaluation transition is provable for u and v is fully evaluated. In the second proof an evaluation transition is provable for v. In the third proof both u and v are fully evaluated. By the induction hypothesis, only one evaluation transition is provable for either u or v and therefore only one evaluation transition is provable for the term $(u\ v)$.

If the outermost function symbol is a conditional expression of the form (if u then v else w) then any provable transition for x will have a proof with one of the following forms:

$$\frac{\dfrac{\vdots}{u \longrightarrow u'}}{\text{if } u \text{ then } v \text{ else } w \longrightarrow \text{if } u' \text{ then } v \text{ else } w}\ (\mathit{if1})$$

$$\frac{u \xrightarrow{k} u' \quad k \neq 0}{\text{if } u \text{ then } v \text{ else } w \longrightarrow v}\ (\mathit{if2}) \qquad \frac{u \xrightarrow{0} u'}{\text{if } u \text{ then } v \text{ else } w \longrightarrow w}\ (\mathit{if3})$$

In the first proof an evaluation transition is provable for u. In the second proof, u is a constant other than 0. In the third proof, u is the constant 0. By the induction hypothesis, only one evaluation transition is provable for u. Therefore only one evaluation transition is provable for the term (if u then v else w).

□

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 an insufficiently abstract, and therefore uninteresting, equivalence on expressions. Although we don't have a full characterization of bisimilarity for our language, the following result shows that at least the worst doesn't happen:

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

Proof Outline: Consider the relation S that relates all terms that are identical up to renaming of bound identifiers. We will show that this relation is a bisimulation. To do this, we need to show that for all expressions x, x' such that $x\ S\ x'$, it follows that for every provable transition $x \xrightarrow{\alpha} z$ there is provable transition $x' \xrightarrow{\alpha} z'$ such that $z\ S\ z'$. The proof is by structural induction on the nesting depth of function symbols in x. See the appendix for the full version of the proof.
□

Finally, bisimilarity is compatible with the constructs of our language (*i.e.* is a congruence). The trick in this proof is identifying the proper relation to be shown a bisimulation. We need to show that for any bisimulation relation R that relates x and y and any context C, we can construct a bisimulation relation R' that relates $C[x]$ and $C[y]$. The lemmas below defines just such a bisimulation relation.

Lemma 2.6 *Let $S : T(\Sigma) \times T(\Sigma)$ be the relation that relates all terms that are identical up to the renaming of bound identifiers and $I : T(\Sigma) \times T(\Sigma)$ the identity relation. For any bisimulation relation R, let R_0 be the transitive closure of $(R \bigcup S \bigcup I)$ and for all i let R_{i+1} be the transitive closure of:*

$$\begin{array}{lrclrcl}
(1) & x\ R_i\ x' & \Longrightarrow & x & R_{i+1} & x' \\
(2) & x\ R_i\ x' & \Longrightarrow & (\mathbf{fn}\ a => x) & R_{i+1} & (\mathbf{fn}\ a => x') \\
(3) & x\ R_i\ x',\ y\ R_i\ y' & \Longrightarrow & x\ y & R_{i+1} & x'\ y' \\
(4) & x\ R_i\ x',\ y\ R_i\ y',\ z\ R_i\ z' & \Longrightarrow & \text{if } x \text{ then } y \text{ else } z & R_{i+1} & \text{if } x' \text{ then } y' \text{ else } z' \\
(5) & y\ R_i\ y' & \Longrightarrow & x[y/a] & R_{i+1} & x[y'/a]
\end{array}$$

For all R_i, w, w', e, a,

$$w\ R_i\ w' \Rightarrow w[e/a]\ R_i\ w'[e/a].$$

Proof Outline:

We will prove this by induction on i. For the base case, $R_0 = R \bigcup S \bigcup I$. By proposition 2.5, S is a bisimulation relation. The identity relation is trivially a bisimulation relation. By hypothesis R is a bisimulation relation. Since the union of bisimulation relations is also a bisimulation relation, $R \cup S \cup I$ is a bisimulation relation. Since by proposition 2.1, $w \xrightarrow{[e/a]} w[e/a]$ and $w' \xrightarrow{[e/a]} w'[e/a]$ we know that $w[e/a]\ R_0\ w'[e/a]$.

For our induction step we will show, if

$$\forall w, w', e, a,\ (w\ R_i\ w' \Rightarrow w[e/a]\ R_i\ w'[e/a]).$$

then

$$\forall w, w', e, a,\ (w\ R_{i+1}\ w' \Rightarrow w[e/a]\ R_{i+1}\ w'[e/a]).$$

We will prove the induction by case analysis of the definition of R_{i+1}. See the appendix for the full version of the proof.
□

Lemma 2.7 *Let $S : T(\Sigma) \times T(\Sigma)$ be the relation that relates all terms that are identical up to the renaming of bound identifiers and $I : T(\Sigma) \times T(\Sigma)$ the identity relation. For any $R : T(\Sigma) \times T(\Sigma)$, let R' be the least relation such that:*

$$
\begin{array}{llcrll}
(1) & x\ I\ x' & \Longrightarrow & x & R' & x' \\
(2) & x\ S\ x' & \Longrightarrow & x & R' & x' \\
(3) & x\ R\ x' & \Longrightarrow & x & R' & x' \\
(4) & x\ R'\ x' & \Longrightarrow & (\mathbf{fn}\ a => x) & R' & (\mathbf{fn}\ a => x') \\
(5) & x\ R'\ x',\ y\ R'\ y' & \Longrightarrow & x\ y & R' & x'\ y' \\
(6) & x\ R'\ x',\ y\ R'\ y',\ z\ R'\ z' & \Longrightarrow & \text{if } x \text{ then } y \text{ else } z & R' & \text{if } x' \text{ then } y' \text{ else } z' \\
(7) & y\ R'\ y' & \Longrightarrow & x[y/a] & R' & x[y'/a] \\
(8) & x\ R'\ x',\ x'\ R'\ x'' & \Longrightarrow & x & R' & x''
\end{array}
$$

If R is a bisimulation relation, then the relation R' is a bisimulation relation.

Proof Outline:

We can can construct R' as $\bigcup R_i$ where R_0 is the transitive closure of $(R \bigcup S \bigcup I)$ and for all i, R_{i+1} is the transitive closure of:

$$
\begin{array}{llcrll}
(1) & x\ R_i\ x' & \Longrightarrow & x & R_{i+1} & x' \\
(2) & x\ R_i\ x' & \Longrightarrow & (\mathbf{fn}\ a => x) & R_{i+1} & (\mathbf{fn}\ a => x') \\
(3) & x\ R_i\ x',\ y\ R_i\ y' & \Longrightarrow & x\ y & R_{i+1} & x'\ y' \\
(4) & x\ R_i\ x',\ y\ R_i\ y',\ z\ R_i\ z' & \Longrightarrow & \text{if } x \text{ then } y \text{ else } z & R_{i+1} & \text{if } x' \text{ then } y' \text{ else } z' \\
(5) & y\ R_i\ y' & \Longrightarrow & x[y/a] & R_{i+1} & x[y'/a]
\end{array}
$$

We would like to show that R' is a bisimulation relation. That is, for all expressions w, w',

$$(w\ R'\ w' \text{ and } w \xrightarrow{\alpha} z) \Rightarrow (\exists z', w' \xrightarrow{\alpha} z' \text{ and } z\ R'\ z')$$

and

$$(w\ R'\ w' \text{ and } w' \xrightarrow{\alpha} z') \Rightarrow (\exists z, w \xrightarrow{\alpha} z \text{ and } z\ R'\ z').$$

We will show this by induction on i. For the base case $R_0 = R \bigcup S \bigcup I$. By proposition 2.5, S is a bisimulation relation. The identity relation is trivially a bisimulation relation. By hypothesis R is a bisimulation relation. Since the union of bisimulation relations is also a bisimulation relation, $R \cup S \cup I$ is a bisimulation relation.

For the induction step, we will show:

if

$$\forall w, w',\ [(w\ R_i\ w' \text{ and } w \xrightarrow{\alpha} z) \Rightarrow (\exists z', w' \xrightarrow{\alpha} z' \text{ and } z\ R'\ z')]$$

then

$$\forall w, w',\ [(w\ R_{i+1}\ w' \text{ and } w \xrightarrow{\alpha} z) \Rightarrow (\exists z', w' \xrightarrow{\alpha} z' \text{ and } z\ R'\ z')].$$

Once we have show this property, the desired result holds immediately. That is, if $w\ R'\ w'$ then by the definition of R' there exists some R_i such that $w\ R_i\ w'$. So if $w \xrightarrow{\alpha} z$ then by the proposition above there exists some z' such that $w' \xrightarrow{\alpha} z'$ and $z\ R'\ z'$.

We will do the proof by considering each case of the definition of R_{i+1} separately. See the appendix for the full version of the proof.
□

Proposition 2.8 *For all contexts $C[\ ]$ and all expressions $x, y \in T(\Sigma)$, if $x \sim y$ then $C[x] \sim C[y]$.*

Proof: We need to show that for any bisimulation relation R that relates x and y and any context C, we can construct a bisimulation relation R' that relates $C[x]$ and $C[y]$. The result follows immediately from the previous lemma.
□

3 Debugging language

In this section, we define our debugger as an extension of the syntax and semantics of the programming language. In particular, we extend the syntax of the programming language with a *focusing operator* ($\langle\!|\ |\!\rangle$) that allows the scope of attention to be focused on the evaluation of a particular subexpression. A *debugging state* consists of an expression from the programming language together with a *debugging context*, which is a list of "coexpressions." A *coexpression*, in essence, is either an ordinary programming language expression that has a designated missing subterm, or else is a *substitution coexpression* indicating that we have moved within the scope of a substitution that has yet to be applied. The coexpression corresponding to the conditional statement (if e_1 then e_2 else e_3) is abbreviated $\{e_1?e_2, e_3\}$.

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

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

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

Any term x in the programming language can be packaged together with an empty debugging context ϵ to form a debugging state $\epsilon\langle\!|x|\!\rangle$. The transition rules for the debugger are defined in such a way that the term x and the term $\epsilon\langle\!|x|\!\rangle$ 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 4. These operations can be viewed as explicit actions taken by the user to modify the debugging state. 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 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

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

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

$$\kappa{:}\{\,\text{-}\ \ y\}\langle\!|(\mathbf{fn}\ a => x)|\!\rangle \overset{\text{fn}}{\Downarrow} \kappa{:}\{y/a'\}\langle\!| x[a'/a] |\!\rangle \qquad \kappa\langle\!|\text{if } x \text{ then } y \text{ else } z|\!\rangle \overset{\text{else}}{\Downarrow} \kappa{:}\{x?y,\ \text{-}\,\}\langle\!| z |\!\rangle$$

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

Figure 4: Focusing rules

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. For clarity, 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 applies 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.

For a simple example of a debugging transition, consider the case where the constant function ($\mathbf{fn}$ a => 1) is applied to the argument 2. From the semantic definition of the programming language we can infer the following transition for the expression:

$$\dfrac{\dfrac{a \xrightarrow{a} a \quad 1 \xrightarrow{[2/a]} a}{(\mathbf{fn}\ a => 1) \xrightarrow{\square 2} 1}\ (tp5)}{(\mathbf{fn}\ a => 1)\ 2 \longrightarrow 1}\ (ap3).$$

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:

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

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

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

Figure 5 defines the 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 - db8). 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).

Figure 6 defines the typing steps for debugging states. As with the programming language typing rules, the debugging language typing rules serve to classify fully evaluated terms. Since the term must be fully evaluated, the typing rules only apply when the debugging context is empty.

Figure 7 defines 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. Rules (ke2) - (ke7) state 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 (sub5) for the programming language.

$$\frac{\kappa \Longrightarrow \kappa'}{\kappa\llparenthesis x \rrparenthesis \Longrightarrow \kappa'\llparenthesis x \rrparenthesis}\ (\mathit{db1}) \qquad \frac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad x \longrightarrow x'}{\kappa\llparenthesis x \rrparenthesis \Longrightarrow \kappa\llparenthesis x' \rrparenthesis}\ (\mathit{db2}) \qquad \frac{\kappa \stackrel{[e/a]}{\Longrightarrow} \kappa' \quad x \stackrel{[e/a]}{\longrightarrow} x'}{\kappa\llparenthesis x \rrparenthesis \Longrightarrow \kappa'\llparenthesis x' \rrparenthesis}\ (\mathit{db3})$$

$$\frac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad x \stackrel{\Box v}{\longrightarrow} x'}{\kappa{:}\{\ \text{-}\ \ v\}\llparenthesis x \rrparenthesis \Longrightarrow \kappa\llparenthesis x' \rrparenthesis}\ (\mathit{db4}) \qquad \frac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad x \longrightarrow x' \quad y \stackrel{v}{\longrightarrow} y'}{\kappa{:}\{x\ \ \text{-}\ \}\llparenthesis y \rrparenthesis \Longrightarrow \kappa{:}\{x'\ \ \text{-}\ \}\llparenthesis y \rrparenthesis}\ (\mathit{db5})$$

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

$$\frac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad x \stackrel{0}{\longrightarrow} x'}{\kappa{:}\{\ \text{-}\ ?y, z\}\llparenthesis x \rrparenthesis \Longrightarrow \kappa\llparenthesis z \rrparenthesis}\ (\mathit{db8}) \qquad \frac{\kappa \stackrel{*}{\Longrightarrow} d}{\kappa\llparenthesis x \rrparenthesis \Longrightarrow d}\ (\mathit{db9})$$

Figure 5: Evaluation rules for debugging states

$$\frac{x \stackrel{k}{\longrightarrow} x'}{\epsilon\llparenthesis x \rrparenthesis \stackrel{k}{\Longrightarrow} \epsilon\llparenthesis x' \rrparenthesis}\ (\mathit{dt1}) \qquad \frac{x \stackrel{a}{\longrightarrow} x'}{\epsilon\llparenthesis x \rrparenthesis \stackrel{a}{\Longrightarrow} \epsilon\llparenthesis x' \rrparenthesis}\ (\mathit{dt2}) \qquad \frac{x \stackrel{\mathsf{v}}{\longrightarrow} x'}{\epsilon\llparenthesis x \rrparenthesis \stackrel{\mathsf{v}}{\Longrightarrow} \epsilon\llparenthesis x' \rrparenthesis}\ (\mathit{dt3}) \qquad \frac{x \stackrel{\Box e}{\longrightarrow} x'}{\epsilon\llparenthesis x \rrparenthesis \stackrel{\Box e}{\Longrightarrow} \epsilon\llparenthesis x' \rrparenthesis}\ (\mathit{dt4})$$

Figure 6: Typing rules for debugging states

The rules in Figure 8 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.

The rules in Figure 9 describe the generation and propagation of control information within a debugging context. These rules (tr1 - tr3) ensure that a program evaluates in the same order when it is being debugged as when it is not being debugged.

The last set of rules (see Figure 10) specifies how substitutions are applied to debugging contexts (see figure 9). 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 - sb7).

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

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

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

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

Figure 7: Evaluation rules for debugging contexts

$$\frac{\kappa \stackrel{*}{\Longrightarrow} d}{\kappa\colon x \stackrel{*}{\Longrightarrow} d}\ (\mathit{br1}) \qquad \frac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad x \stackrel{0}{\longrightarrow} x'}{\kappa\colon \{x?\ \text{-}\ ,z\} \stackrel{*}{\Longrightarrow} \kappa\llparenthesis z\rrparenthesis}\ (\mathit{br2}) \qquad \frac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad x \stackrel{k}{\longrightarrow} x' \quad k \neq 0}{\kappa\colon \{x?y,\ \text{-}\ \} \stackrel{*}{\Longrightarrow} \kappa\llparenthesis y\rrparenthesis}\ (\mathit{br3})$$

Figure 8: Conditional branching rules for debugging contexts

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.

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

Proof: 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. This means that none of these rules can be used to draw inferences about pure programming language expressions.
□

$$\epsilon \xRightarrow{!} \epsilon \quad (tr0) \qquad \frac{\kappa \xRightarrow{!} \kappa' \quad y \xrightarrow{\text{v}} y'}{\kappa\colon\{\,\text{-}\ \ y\} \xRightarrow{!} \kappa\colon\{\,\text{-}\ \ y\}} \quad (tr1)$$

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

Figure 9: Trigger rules for debugging contexts

$$\frac{\kappa \xRightarrow{!} \kappa'}{\kappa\colon\{v/a\} \xRightarrow{[v/a]} \kappa} \quad (sb1) \qquad \frac{\kappa \xRightarrow{[e/a]} \kappa' \quad x \xrightarrow{[e/a]} x'}{\kappa\colon\{\,\text{-}\ \ x\} \xRightarrow{[e/a]} \kappa'\colon\{\,\text{-}\ \ x'\}} \quad (sb2) \qquad \frac{\kappa \xRightarrow{[e/a]} \kappa' \quad x \xrightarrow{[e/a]} x'}{\kappa\colon\{x\ \ \text{-}\,\} \xRightarrow{[e/a]} \kappa'\colon\{x'\ \ \text{-}\,\}} \quad (sb3)$$

$$\frac{\kappa \xRightarrow{[e/a]} \kappa' \quad y \xrightarrow{[e/a]} y' \quad z \xrightarrow{[e/a]} z'}{\kappa\colon\{\,\text{-}\ ?y, z\} \xRightarrow{[e/a]} \kappa'\colon\{\,\text{-}\ ?y', z'\}} \quad (sb4) \qquad \frac{\kappa \xRightarrow{[e/a]} \kappa' \quad x \xrightarrow{[e/a]} x' \quad z \xrightarrow{[e/a]} z'}{\kappa\colon\{x?\ \text{-}\ , z\} \xRightarrow{[e/a]} \kappa'\colon\{x'?\ \text{-}\ , z'\}} \quad (sb5)$$

$$\frac{\kappa \xRightarrow{[e/a]} \kappa' \quad x \xrightarrow{[e/a]} x' \quad y \xrightarrow{[e/a]} y'}{\kappa\colon\{x?y,\ \text{-}\,\} \xRightarrow{[e/a]} \kappa'\colon\{x'?y',\ \text{-}\,\}} \quad (sb6) \qquad \frac{\kappa \xRightarrow{[e/a]} \kappa' \quad x \xrightarrow{[e/a]} x' \quad a \neq b}{\kappa\colon\{x/b\} \xRightarrow{[e/a]} \kappa'\colon\{x'/b\}} \quad (sb7)$$

Figure 10: Substitution rules for debugging contexts

The second result states that bisimilar expressions placed in the same debugging context yield bisimilar debugging states.

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

Proof Outline: Any proof of a transition of $\kappa\langle\!|x|\!\rangle$, will have an evaluation rule or a typing rule as the last step, since those are the only rules with the debugging state function symbol on the left-hand side of the conclusion. We will do the proof by case analysis on the last rule in the proof of the transition. For the full version of the proof, see the appendix.
□

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 substitution transitions.*

Proof: For any proof of a transition of $\epsilon\langle\!|x|\!\rangle$ the last step in the proof must be by (db2), (dt1), (dt2), (dt3) or (dt4). In all five cases the result is immediate.
□

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 substitution transitions) to the original one. The result is proved by a case analysis of the possible focusing operations and transitions.

Theorem 3.1 *For all debugging states d and d', if $d \overset{\delta}{\Downarrow} d'$, then d is bisimilar to d', excluding substitution transitions.*

Proof Outline: Let S relate debugging terms $\kappa\langle\!|x|\!\rangle$ and $\kappa\langle\!|x'|\!\rangle$ iff x and x' are identical up to the renaming of bound identifiers. Let I be the identity relation. Let R be the transitive closure of $S \cup I \cup \{(d, d') : d \overset{\delta}{\Downarrow} d'\}$. We will show that R' is a bisimulation relation. Since S and I are bisimulation relations, it is sufficient to show $\{(d, d') : d \overset{\delta}{\Downarrow} d'\}$ is a bisimulation relation. By inspection of the focusing rules, we know that the programming language expression in focus must be an application, function definition or conditional expression. Since the only such expression that can do a typing rule is a function definition and from the focusing rule we know in this case the debugging context is not empty, therefore the last rule in a proof of a transition of d, cannot be a typing rule and must in fact be an evaluation rule. Similarly, the debugging context for d' cannot be empty, therefore the last rule in a proof of a transition of d', cannot be a typing rule and must in fact be an evaluation rule. We will do the proof by a case analysis on the last rule in the proof of the possible transitions. For the full version of the proof, see the appendix.
□

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. 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

[App92] Andrew W. Appel. *Compiling with Continuatations.* Cambridge University Press, Cambridge, 1992.

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

[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.

[GV92] Jan Friso Groote and Frits Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100:202–260, 1992.

[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.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[Par81] 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.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.

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

[Tol92] Andrew Tolmach. *Debugging Standard ML.* PhD thesis, Princeton University, October 1992.

A Proposition 2.1

For all expressions x, x' all expressions e, and all identifiers a:

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

Proof: The proof is by structural induction on x. If x is an arbitrary constant k, then the transition $k \xrightarrow{[e/a]} k$ is the only transition, labeled with $[e/a]$, provable by rule (sub1). Since no other substitution rule has a constant as a function symbol on the left-hand side of the conclusion, no other transitions are provable for k. Since $k[e/a] = k$, it follows that for all constants k,

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

If x is an identifier then we need to consider both the case where $x = a$ and the case when $x \neq a$. If $x = a$ then the transition $a \xrightarrow{[e/a]} e$ is the only transition, labeled with $[e/a]$, provable by the rule (sub2). Since no other rules have the function symbol a on the left-hand side of the conclusion and $a[e/a] = e$, we have for all identifiers a,

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

If x is an arbitrary identifier b, where $b \neq a$, then the transition $b \xrightarrow{[e/a]} b$ is provable by the rule (sub3). Since no other rules have the function symbol b on the left-hand side of the conclusion and $b[e/a] = b$, we have for all identifiers a, b such that $a \neq b$,

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

If the outermost function symbol of x is an application then x is of the form $(u\ v)$. Since (sub4) is the only rule with the application function symbol on the left-hand side of the conclusion, any proof of a transition of x labeled with $[e/a]$ will be of the form:

$$\frac{\dfrac{\vdots}{u \xrightarrow{[e/a]} u'} \quad \dfrac{\vdots}{v \xrightarrow{[e/a]} v'}}{u\ v \xrightarrow{[e/a]} u'\ v'} \ (sub4) \quad .$$

By the induction hypothesis, $(u \xrightarrow{[e/a]} u' \iff u' = u[e/a])$ and $(v \xrightarrow{[e/a]} v' \iff v' = v[e/a])$. Since $(u[e/a]\ v[e/a]) = (u\ v)[e/a]$,

$$u\ v \xrightarrow{[e/a]} x' \iff x' = (u\ v)[e/a].$$

If the outermost function symbol is a function definition then x is of the form $(\mathbf{fn}\ b => u)$ and we need to consider both the case where $b = a$ and the case where $b \neq a$. If $b = a$ then any proof of a transition of x labeled with $[e/a]$ will be of the form:

$$\frac{a \xrightarrow{a} a}{(\mathbf{fn}\ a => u) \xrightarrow{[e/a]} (\mathbf{fn}\ a => u)} \ (sub6) \quad .$$

Since $(\mathbf{fn}\ a => u)[e/a] = (\mathbf{fn}\ a => u)$,

$$(\mathbf{fn}\ a => u) \xrightarrow{[e/a]} x' \iff x' = (\mathbf{fn}\ a => u)[e/a].$$

If $b \neq a$ then any proof of a transition of x labeled with $[e/a]$ will be of the form:

$$\frac{b \xrightarrow{b} b \quad \dfrac{\vdots}{u \xrightarrow{[b'/b]} u'} \quad \dfrac{\vdots}{u' \xrightarrow{[e/a]} u''}}{(\mathbf{fn}\ b => u) \xrightarrow{[e/a]} (\mathbf{fn}\ b' => u'')} \quad (sub5)$$
,

where b' is the first identifier name that does not occur in either $(\mathbf{fn}\ b => u)$ or e. By the induction hypothesis, $(u \xrightarrow{[b/b']} u' \iff u' = u[b/b'])$. Since b and b' are both identifiers, the nesting depth of function symbols is the same in u and $u[b'/b]$. Therefore it follows by the induction hypothesis, $(u' \xrightarrow{[e/a]} u'' \iff u'' = u'[e/a])$. So if $b \neq a$, we have $(\mathbf{fn}\ b => u) \xrightarrow{[e/a]} (\mathbf{fn}\ b' => u[b'/b][e/a])$. Since $(\mathbf{fn}\ b => u)[e/a] = (\mathbf{fn}\ b' => u[b'/b][e/a])$, we have shown that for all identifiers b,

$$(\mathbf{fn}\ b => u) \xrightarrow{[e/a]} x' \iff x' = (\mathbf{fn}\ b => u)[e/a].$$

If the outermost function symbol is a conditional expression, then x is of the form (if u then v else w) and any proof of a transition of x labeled with $[e/a]$ will be of the form:

$$\frac{\dfrac{\vdots}{u \xrightarrow{[e/a]} u'} \quad \dfrac{\vdots}{v \xrightarrow{[e/a]} v'} \quad \dfrac{\vdots}{w \xrightarrow{[e/a]} w'}}{\text{if } u \text{ then } v \text{ else } w \xrightarrow{[e/a]} \text{if } u' \text{ then } v' \text{ else } w'} \quad (sub7)$$
.

By the induction hypothesis $(u \xrightarrow{[e/a]} u' \iff u' = u[e/a])$, $(v \xrightarrow{[e/a]} v' \iff v' = v[e/a])$ and $(w \xrightarrow{[e/a]} w' \iff w' = w[e/a])$. Therefore, since (if $u[e/a]$ then $v[e/a]$ else $w[e/a]$) $=$ (if u then v else w)$[e/a]$, we have:

$$(\text{if } u \text{ then } v \text{ else } w) \xrightarrow{[e/a]} x' \iff x' = (\text{if } u \text{ then } v \text{ else } w)[e/a].$$

□

B Proposition 2.5

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

Proof

Consider the relation S that relates all terms that are identical up to renaming of bound identifiers. We will show that this relation is a bisimulation. Since the relation S is symmetric it is sufficient to show that for all expressions x, x' such that $x\ S\ x'$, it follows that for every provable transition $x \xrightarrow{\alpha} z$ there is a provable transition $x' \xrightarrow{\alpha} z'$ such that $z\ S\ z'$. The proof is by structural induction on the nesting depth of function symbols in x.

In the base case, x is a constant or an identifier and $x\ S\ x' \iff x = x'$. The result is immediate in this case. In order to prove the induction step, we will do a case analysis on the possible proofs for a transition $x \xrightarrow{\alpha} z$. Since the outermost function symbol in x is a function definition, an application or a conditional expression, it follows we need to consider the cases where the last rule in the proof has one of these function symbols on the left-hand side of the conclusion. The relevant rules are: (tp4), (tp5), (sub4), (sub5), (sub6), (sub7), (ap1), (ap2), (ap3), (if1), (if2) and (if3).

Case 1: (tp4) If the last rule in the proof of the transition is (tp4), then the proof is of the form:

$$(\mathbf{fn}\ a \Rightarrow u) \xrightarrow{\mathrm{v}} (\mathbf{fn}\ a \Rightarrow u) \quad (tp4),$$

where $x = (\mathbf{fn}\ a \Rightarrow u)$ and $x' = (\mathbf{fn}\ a' \Rightarrow u')$. It follows immediately that:

$$(\mathbf{fn}\ a' \Rightarrow u') \xrightarrow{\mathrm{v}} (\mathbf{fn}\ a' \Rightarrow u') \quad (tp4),$$

where by assumption $(\mathbf{fn}\ a \Rightarrow u)\ S\ (\mathbf{fn}\ a' \Rightarrow u')$.

Case 2: (tp5) If the last rule in the proof of the transition is (tp5), then the proof is of the form:

$$\frac{a \xrightarrow{a} a \qquad \dfrac{\vdots}{u \xrightarrow{[e/a]} v}}{(\mathbf{fn}\ a \Rightarrow u) \xrightarrow{\Box e} v} \quad (tp5),$$

where $x = (\mathbf{fn}\ a \Rightarrow u)$ and $x' = (\mathbf{fn}\ a' \Rightarrow u')$. By proposition 2.1, $v = u[e/a]$ and the transition $u' \xrightarrow{[e/a']} u'[e/a']$ is provable. Since by assumption $(\mathbf{fn}\ a \Rightarrow u)\ S\ (\mathbf{fn}\ a' \Rightarrow u')$ we know that $u[a'/a]\ S\ u'$ and $u[a'/a][e/a']\ S\ u'[e/a']$. Since $(\mathbf{fn}\ a \Rightarrow u)\ S\ (\mathbf{fn}\ a' \Rightarrow u')$ the terms have the same free identifiers and a' does not occur free in either u or u'. Since $u[a'/a][e/a']\ S\ u[e/a]$, we have that $u[e/a]\ S\ u'[e/a']$. Therefore we have:

$$\frac{a' \xrightarrow{a'} a' \qquad \dfrac{\vdots}{u' \xrightarrow{[e/a']} v'}}{(\mathbf{fn}\ a' \Rightarrow u') \xrightarrow{\Box e} v'} \quad (tp5)$$

where $v\ S\ v'$.

Case 3: (sub4) If the last rule in the proof of the transition is (sub4), then the transition is of the form $(u\ v) \xrightarrow{[e/a]} z$ where $x = (u\ v)$, $x' = (u'\ v')$ and by proposition 2.1, $z = (u\ v)[e/a]$. By proposition 2.1, there is a proof that $(u'\ v') \xrightarrow{[e/a]} (u'\ v')[e/a]$ where by the definition of S,

$(u\ v)[e/a]\ S\ (u'\ v')[e/a]$.

Case 4: (sub5),(sub6) If the last rule in the proof of the transition is (sub5) or (sub6), then the transition is of the form $(\mathbf{fn}\ a => u) \xrightarrow{[e/b]} z$, where $x = (\mathbf{fn}\ a => u)$, $x' = (\mathbf{fn}\ a' => u')$ and by proposition 2.1, $z = (\mathbf{fn}\ a => u)[e/b]$. By proposition 2.1, there is a proof that $(\mathbf{fn}\ a' => u') \xrightarrow{[e/b]} (\mathbf{fn}\ a' => u')[e/b]$ where by the definition and properties of S, $(\mathbf{fn}\ a => u)[e/b]\ S\ (\mathbf{fn}\ a' => u')[e/b]$.

Case 5: (sub7) If the last rule in the proof of the transition is (sub7), then the transition is of the form: $(\text{if } u \text{ then } v \text{ else } w) \xrightarrow{[e/a]} z$ where $x = (\text{if } u \text{ then } v \text{ else } w)$, $x' = (\text{if } u' \text{ then } v' \text{ else } w')$ and by proposition 2.1, $z = (\text{if } u \text{ then } v \text{ else } w)[e/a]$. By proposition 2.1, there is a proof that $(\text{if } u' \text{ then } v' \text{ else } w') \xrightarrow{[e/a]} (\text{if } u' \text{ then } v' \text{ else } w')[e/a]$ where by the definition of S, $(\text{if } u \text{ then } v \text{ else } w)[e/a]\ S\ (\text{if } u' \text{ then } v' \text{ else } w')[e/a]$.

Case 6: (ap1) If the last rule in the proof of the transition is (ap1), then the proof is of the form:

$$\frac{\dfrac{\vdots}{u \longrightarrow y} \quad v \xrightarrow{\mathrm{v}} z}{u\ v \longrightarrow y\ v}\ (ap1),$$

where $x = (u\ v)$ and $x' = (u'\ v')$. By the induction hypothesis, there exists y' and z' where there are proofs of the transitions $u' \longrightarrow y'$ and $v' \xrightarrow{\mathrm{v}} z'$ such that $y\ S\ y'$. Therefore there is a proof:

$$\frac{\dfrac{\vdots}{u' \longrightarrow y'} \quad v' \xrightarrow{\mathrm{v}} z'}{u'\ v' \longrightarrow y'\ v'}\ (ap1),$$

where from the definition of S, $(y\ v)\ S\ (y'\ v')$.

Case 7: (ap2) If the last rule in the proof of the transition is (ap2), then the proof is of the form:

$$\frac{\dfrac{\vdots}{v \longrightarrow z}}{u\ v \longrightarrow u\ z}\ (ap2),$$

where $x = (u\ v)$ and $x' = (u'\ v')$. By the induction hypothesis, there exists z' where there exists a proof of $v' \longrightarrow z'$ such that $z\ S\ z'$. Therefore there is a proof:

$$\frac{\dfrac{\vdots}{v' \longrightarrow z'}}{u'\ v' \longrightarrow u'\ z'}\ (ap2),$$

where from the definition of S, $(u\ z)\ S\ (u'\ z')$.

Case 8: (ap3) If the last rule in the proof of the transition is (ap3), then the proof is of the form:

$$\frac{\dfrac{\vdots}{u \xrightarrow{\Box v} z}}{u\ v \longrightarrow z}\ (\mathit{ap3}),$$

where $x = (u\ v)$ and $x' = (u'\ v')$. Since the only rule that can prove a "$\Box$" transition is (tp5), we know the proof of $u \xrightarrow{\Box v} z$ is of the form:

$$\frac{a \xrightarrow{a} a \quad \dfrac{\vdots}{w \xrightarrow{[v/a]} z}}{(\textbf{fn}\ a => w) \xrightarrow{\Box v} z}\ (\mathit{tp5}),$$

where $u = (\textbf{fn}\ a => w)$ and by proposition 2.1, $z = w[v/a]$. Since $u\ S\ u'$, it follows that $u' = (\textbf{fn}\ a' => w')$ and $w[a'/a]\ S\ w'$. By proposition 2.1, there is a proof of the transition $w' \xrightarrow{[v'/a']} w'[v'/a']$. By the definition of S and since $v\ S\ v'$ and a' is not free in w' and hence also w, $w[a'/a][v/a']\ S\ w'[v'/a']$ where $w[a'/a][v/a']\ S\ w[v/a]$. Therefore:

$$\frac{\dfrac{a' \xrightarrow{a'} a' \quad \dfrac{\vdots}{w' \xrightarrow{[v'/a']} w'[v'/a']}}{(\textbf{fn}\ a' => w') \xrightarrow{\Box v'} w'[v'/a']}\ (\mathit{tp5})}{(\textbf{fn}\ a' => w')\ v' \longrightarrow w'[v'/a']}\ (\mathit{ap3}).$$

where $w'[v'/a']\ S\ z$.

Case 9: (if1) If the last rule in the proof of the transition is (if1), then the proof is of the form:

$$\frac{\dfrac{\vdots}{u \longrightarrow z}}{\text{if } u \text{ then } v \text{ else } w \longrightarrow \text{if } z \text{ then } v \text{ else } w}\ (\mathit{if1}),$$

where $x = (\text{if } u \text{ then } v \text{ else } w)$ and $x' = (\text{if } u' \text{ then } v' \text{ else } w')$. By the induction hypothesis, there exists some z' such that $u' \longrightarrow z'$ is provable and $z\ S\ z'$. Therefore there is a proof:

$$\frac{\dfrac{\vdots}{u' \longrightarrow z'}}{\text{if } u' \text{ then } v' \text{ else } w' \longrightarrow \text{if } z' \text{ then } v' \text{ else } w'}\ (\mathit{if1}),$$

where by the definition of S, $(\text{if } z \text{ then } v \text{ else } w)\ S\ (\text{if } z' \text{ then } v' \text{ else } w')$.

Case 10: (if2) If the last rule in the proof of the transition is (if2), then the proof is of the form:

$$\frac{u \xrightarrow{k} u \quad k \neq 0}{\text{if } u \text{ then } v \text{ else } w \longrightarrow v}\ (\mathit{if2}),$$

where $x =$ (if u then v else w) and $x' =$ (if u' then v' else w'). By the induction hypothesis, there is a proof that $u' \xrightarrow{k} u'$. So there is a proof:

$$\frac{u' \xrightarrow{k} u' \quad k \neq 0}{\text{if } u' \text{ then } v' \text{ else } w' \longrightarrow v'} \; (\mathit{if2}),$$

where $v \; S \; v'$.

Case 11: (if3) If the last rule in the proof of the transition is (if3), then the proof is of the form:

$$\frac{u \xrightarrow{0} u}{\text{if } u \text{ then } v \text{ else } w \longrightarrow w} \; (\mathit{if3}),$$

where $x =$ (if u then v else w) and $x' =$ (if u' then v' else w'). By the induction hypothesis, there is a proof that $u' \xrightarrow{0} u'$. So there is a proof:

$$\frac{u' \xrightarrow{0} u'}{\text{if } u' \text{ then } v' \text{ else } w' \longrightarrow w'} \; (\mathit{if3}),$$

where $w \; S \; w'$.
□

C Lemma 2.6

Let $S : T(\Sigma) \times T(\Sigma)$ be the relation that relates all terms that are identical up to the renaming of bound identifiers and $I : T(\Sigma) \times T(\Sigma)$ the identity relation. For any bisimulation relation R, let R_0 be the transitive closure of $(R \bigcup S \bigcup I)$ and for all i let R_{i+1} be the transitive closure of:

$$\begin{array}{llcrll}
(1) & x \; R_i \; x' & \Longrightarrow & x & R_{i+1} & x' \\
(2) & x \; R_i \; x' & \Longrightarrow & (\mathbf{fn}\; a => x) & R_{i+1} & (\mathbf{fn}\; a => x') \\
(3) & x \; R_i \; x',\; y \; R_i \; y' & \Longrightarrow & x\; y & R_{i+1} & x'\; y' \\
(4) & x \; R_i \; x',\; y \; R_i \; y',\; z \; R_i \; z' & \Longrightarrow & \text{if } x \text{ then } y \text{ else } z & R_{i+1} & \text{if } x' \text{ then } y' \text{ else } z' \\
(5) & y \; R_i \; y' & \Longrightarrow & x[y/a] & R_{i+1} & x[y'/a]
\end{array}$$

For all R_i, w, w', e, a,

$$w \; R_i \; w' \Rightarrow w[e/a] \; R_i \; w'[e/a].$$

Proof:

We will prove this by induction on i. For the base case $R_0 = R \bigcup S \bigcup I$. By proposition 2.5, S is a bisimulation relation. The identity relation is trivially a bisimulation relation. By hypothesis R is a bisimulation relation. Since the union of bisimulation relations is also a bisimulation relation, $R \cup S \cup I$ is a bisimulation relation. Since by proposition 2.1, $w \xrightarrow{[e/a]} w[e/a]$ and $w' \xrightarrow{[e/a]} w'[e/a]$ we know that $w[e/a] \; R_0 \; w'[e/a]$.

For our induction hypothesis, we will assume

$$\forall w, w', e, a,\ (w\ R_i\ w' \Rightarrow w[e/a]\ R_i\ w'[e/a]),$$

and show

$$\forall w, w', e, a,\ (w\ R_{i+1}\ w' \Rightarrow w[e/a]\ R_{i+1}\ w'[e/a]).$$

We will prove the induction by case analysis of the definition of R_{i+1}
Case 1: $w = x$ and $w' = x'$ where $x\ R_i\ x'$

The result follows immediately from the induction hypothesis.

Case 2: $w = (\mathbf{fn}\ b => x)$ and $w' = (\mathbf{fn}\ b => x')$ where $x\ R_i\ x'$

In this case, we need to consider both when $a = b$ and when $a \neq b$. If $a = b$ then $(\mathbf{fn}\ b => x)[e/a] = (\mathbf{fn}\ b => x)$ and $(\mathbf{fn}\ b => x')[e/a] = (\mathbf{fn}\ b => x')$ and the result is immediate.

If $a \neq b$ then $(\mathbf{fn}\ b => x)[e/a] = (\mathbf{fn}\ c => x[c/b][e/a])$ and $(\mathbf{fn}\ b => x')[e/a] = (\mathbf{fn}\ c' => x[c'/b][e/a])$ where c is the first identifier that does not occur in $(\mathbf{fn}\ b => x)$ or e and c' is the first identifier that does not occur in $(\mathbf{fn}\ b => x')$ or e. Let d be an identifier that does not occur in $(\mathbf{fn}\ a => x)$, $(\mathbf{fn}\ a => x')$, or e. By the induction hypothesis, $x[d/b]\ R_i\ x'[d/b]$. And again by the induction hypothesis, $x[d/b][e/a]\ R_i\ x'[d/b][e/a]$. By case 2 of of the definition of R_{i+1}, $(\mathbf{fn}\ d => x[d/b][e/a])\ R_{i+1}\ (\mathbf{fn}\ d => x'[d/b][e/a])$. By changing bound variables and since d does not occur in e, we have:

$$(\mathbf{fn}\ d => x[d/b][e/a])\ S\ (\mathbf{fn}\ c => x[d/b][e/a][c/d])$$

$$S\ (\mathbf{fn}\ c => x[d/b][c/d][e/a])\ S\ (\mathbf{fn}\ c => x[c/b][e/a])$$

and

$$(\mathbf{fn}\ d => x'[d/b][e/a])\ S\ (\mathbf{fn}\ c' => x'[d/b][e/a][c'/d])$$

$$S\ (\mathbf{fn}\ c' => x'[d/b][c'/d][e/a])\ S\ (\mathbf{fn}\ c' => x'[c'/b][e/a]).$$

Therefore we have $(\mathbf{fn}\ c => x)[e/a]\ R_{i+1}\ (\mathbf{fn}\ c => x')[e/a]$.

Case 3: $w = x\ y$ and $w' = x'\ y'$ where $x\ R_i\ x'$ and $y\ R_i\ y'$

In this case $(x\ y)[e/a] = x[e/a]\ y[e/a]$ and $(x'\ y')[e/a] = x'[e/a]\ y'[e/a]$. By the induction hypothesis $x[e/a]\ R_i\ x'[e/a]$ and $y[e/a]\ R_i\ y'[e/a]$. Therefore by case 3 of the definition of R_{i+1}, $(x\ y)[e/a]\ R_{i+1}\ (x'\ y')[e/a]$.

Case 4: $w =$ if x then y else z and $w' =$ if x' then y' else z' where $x\ R_i\ x'$, $y\ R_i\ y'$ and $z\ R_i\ z'$

In this case,

$$(\text{if } x \text{ then } y \text{ else } z)[e/a] = (\text{if } x[e/a] \text{ then } y[e/a] \text{ else } z[e/a])$$

and

$$(\text{if } x' \text{ then } y' \text{ else } z')[e/a] = (\text{if } x'[e/a] \text{ then } y'[e/a] \text{ else } z'[e/a]).$$

By the induction hypothesis $x[e/a]\ R_i\ x'[e/a]$, $y[e/a]\ R_i\ y'[e/a]$ and $z[e/a]\ R_i\ z'[e/a]$. Therefore by case 4 of the definition of R_{i+1},

$$(\text{if } x \text{ then } y \text{ else } z)[e/a]\ R_{i+1}\ (\text{if } x' \text{ then } y' \text{ else } z')[e/a].$$

Case 5: $w = x[y/b]$ and $w' = x[y'/b]$ where $y\ R_i\ y'$

We need to consider both the case where $a = b$ and where $a \neq b$. If $a = b$ then $x[y/b][e/a] = x[y[e/b]/b]$ and $x[y'/b][e/a] = x[y'[e/b]/b]$. By the induction hypothesis $y[e/b]\ R_i\ y'[e/b]$ therefore by case 5 of the definition of R_{i+1}, $x[y[e/b]/b]\ R_{i+1}\ x[y'[e/b]/b]$.

If $a \neq b$, then by the induction hypothesis $y[e/a]\ R_i\ y'[e/a]$. Let d be a fresh identifier. By case 5 of the definition of R_{i+1}, $x[d/b][e/a][y[e/a]/d]\ R_{i+1}\ x[d/b][e/a][y'[e/a]/d]$. By commuting the substitutions we get

$$x[d/b][e/a][y[e/a]/d]\ S\ x[d/b][y/d][e/a]\ S\ x[y/b][e/a]$$

and similarly

$$x[d/b][e/a][y'[e/a]/d]\ S\ x[d/b][y'/d][e/a]\ S\ x[y'/b][e/a].$$

Therefore we have that $x[y/b][e/a]\ R_{i+1}\ x[y'/b][e/a]$.

□

D Lemma 2.7

Let $S : T(\Sigma)\times T(\Sigma)$ be the relation that relates all terms that are identical up to the renaming of bound identifiers and $I : T(\Sigma) \times T(\Sigma)$ the identity relation. For any $R : T(\Sigma) \times T(\Sigma)$, let R' be the least relation such that:

$$\begin{array}{lccrcl}
(1) & x\ I\ x' & \Longrightarrow & x & R' & x' \\
(2) & x\ S\ x' & \Longrightarrow & x & R' & x' \\
(3) & x\ R\ x' & \Longrightarrow & x & R' & x' \\
(4) & x\ R'\ x' & \Longrightarrow & (\mathbf{fn}\ a => x) & R' & (\mathbf{fn}\ a => x') \\
(5) & x\ R'\ x',\ y\ R'\ y' & \Longrightarrow & x\ y & R' & x'\ y' \\
(6) & x\ R'\ x',\ y\ R'\ y',\ z\ R'\ z' & \Longrightarrow & \text{if } x \text{ then } y \text{ else } z & R' & \text{if } x' \text{ then } y' \text{ else } z' \\
(7) & y\ R'\ y' & \Longrightarrow & x[y/a] & R' & x[y'/a] \\
(8) & x\ R'\ x',\ x'\ R'\ x'' & \Longrightarrow & x & R' & x''
\end{array}$$

If R is a bisimulation relation, then the relation R' is a bisimulation relation.

Proof:

We can can construct R' as $\bigcup R_i$ where R_0 is the transitive closure of $(R \bigcup S \bigcup I)$ and for all i, R_{i+1} is the transitive closure of:

$$\begin{array}{lccrcl}
(1) & x\ R_i\ x' & \Longrightarrow & x & R_{i+1} & x' \\
(2) & x\ R_i\ x' & \Longrightarrow & (\mathbf{fn}\ a => x) & R_{i+1} & (\mathbf{fn}\ a => x') \\
(3) & x\ R_i\ x',\ y\ R_i\ y' & \Longrightarrow & x\ y & R_{i+1} & x'\ y' \\
(4) & x\ R_i\ x',\ y\ R_i\ y',\ z\ R_i\ z' & \Longrightarrow & \text{if } x \text{ then } y \text{ else } z & R_{i+1} & \text{if } x' \text{ then } y' \text{ else } z' \\
(5) & y\ R_i\ y' & \Longrightarrow & x[y/a] & R_{i+1} & x[y'/a]
\end{array}$$

We would like to show that R' is a bisimulation relation. That is, for all expressions w, w',

$$(w\ R'\ w' \text{ and } w \xrightarrow{\alpha} z) \Rightarrow (\exists z', w' \xrightarrow{\alpha} z' \text{ and } z\ R'\ z')$$

and

$$(w\ R'\ w' \text{ and } w' \xrightarrow{\alpha} z') \Rightarrow (\exists z, w \xrightarrow{\alpha} z \text{ and } z\ R'\ z').$$

We will show this by induction on i. For the base case $R_0 = R \bigcup S \bigcup I$. By proposition 2.5, S is a bisimulation relation. The identity relation is trivially a bisimulation relation. By hypothesis R is a bisimulation relation. Since the union of bisimulation relations is also a bisimulation relation, $R \cup S \cup I$ is a bisimulation relation.

For the induction step, we will show:

if

$$\forall w, w',\ [(w\ R_i\ w' \text{ and } w \xrightarrow{\alpha} z) \Rightarrow (\exists z', w' \xrightarrow{\alpha} z' \text{ and } z\ R'\ z')]$$

then

$$\forall w, w',\ [(w\ R_{i+1}\ w' \text{ and } w \xrightarrow{\alpha} z) \Rightarrow (\exists z', w' \xrightarrow{\alpha} z' \text{ and } z\ R'\ z')].$$

Once we have show this property, the desired result holds immediately. That is, if $w\ R'\ w'$ then by the definition of R' there exists some R_i such that $w\ R_i\ w'$. So if $w \xrightarrow{\alpha} z$ then by the proposition above there exists some z' such that $w' \xrightarrow{\alpha} z'$ and $z\ R'\ z'$.

We will do the proof by considering each case of the definition of R_{i+1} separately.

Case 1: $w = x$ and $w' = x'$ where $x\ R_i\ x'$

By the induction hypothesis, $w\ R_i\ w'$.

Case 2: $w = (\textbf{fn}\ b => x)$ and $w' = (\textbf{fn}\ b => x')$ where $x\ R_i\ x'$

For any provable transition of $(\textbf{fn}\ a => x)$, the last rule in the proof is (sub5), (sub6), (tp4) or (tp5). We will consider each form of the proof as a separate case.

Case 2a: (sub5)(sub6) If last rule is (sub5) or (sub6) then by proposition 2.1 $z = w[e/a]$ and $z' = w'[e/a]$ and by the previous lemma $z\ R_{i+1}\ z'$.

Case 2b: (tp4) If (tp4) is the last rule, then the proof of the transition of $(\textbf{fn}\ a => x)$ is of the form:

$$(\textbf{fn}\ a => x) \xrightarrow{\mathrm{v}} (\textbf{fn}\ a => x) \quad (tp4).$$

It follows immediately that:

$$(\textbf{fn}\ a => x') \xrightarrow{\mathrm{v}} (\textbf{fn}\ a => x') \quad (tp4),$$

where $(\textbf{fn}\ a => x)\ R_{i+1}\ (\textbf{fn}\ a => x')$ by hypothesis.

Case 2c: (tp5) If (tp5) is the last rule, then the proof of the transition of (**fn** a => x) is of the form:

$$\frac{a \xrightarrow{a} a \quad \dfrac{\vdots}{x \xrightarrow{[e/a]} y}}{(\textbf{fn}\ a => x) \xrightarrow{\Box e} y}\ (tp5).$$

Since $x\ R_i\ x'$, it follows from the previous lemma and proposition 2.1 that $x' \xrightarrow{[e/a]} y'$ is provable where $y\ R_i\ y'$. Therefore we have:

$$\frac{a \xrightarrow{a} a \quad \dfrac{\vdots}{x' \xrightarrow{[e/a]} y'}}{(\textbf{fn}\ a => x') \xrightarrow{\Box e} y'}\ (tp5),$$

where $y\ R_i\ y'$.

Case 3: $w = x\ y$ and $w' = x'\ y'$ where $x\ R_i\ x'$ and $y\ R_i\ y'$

For any provable transition of ($x\ y$), the last rule in the proof is (sub4), (ap1), (ap2), or (ap3).

Case 3a: (sub4) If last rule is (sub4) then by proposition 2.1 $z = w[e/a]$ and $z' = w'[e/a]$ and by the previous lemma $z\ R_{i+1}\ z'$.

Case 3b: (ap1) If the last rule in the proof is (ap1) then the proof is of the form:

$$\frac{\dfrac{\vdots}{x \longrightarrow u} \quad y \xrightarrow{\mathrm{v}} y}{x\ y \longrightarrow u\ y}\ (ap1)$$

By the induction hypothesis, $x' \longrightarrow u'$ such that $u\ R'\ u'$ and $y' \xrightarrow{v} y'$. Therefore we have:

$$\frac{\dfrac{\vdots}{x' \longrightarrow u'} \quad y' \xrightarrow{\mathrm{v}} y'}{x'\ y' \longrightarrow u'\ y'}\ (ap1),$$

where, by case 5 of the definition of R', $(u\ y)\ R'\ (u'\ y')$.

Case 3c: (ap2) If the last rule in the proof is (ap2) then the proof is of the form:

$$\frac{\dfrac{\vdots}{y \longrightarrow u}}{x\ y \longrightarrow x\ u}\ (ap2).$$

By the induction hypothesis, $y' \longrightarrow u'$ such that $u\ R'\ u'$. Therefore we have:

$$\dfrac{\dfrac{\vdots}{y' \longrightarrow u'}}{x'\ y' \longrightarrow x'\ u'}\ (ap2),$$

where by case 5 of the definition of R', $(x\ u)\ R'\ (x'\ u')$.

Case 3d: (ap3) If the last rule in the proof is (ap3) then the proof is of the form:

$$\dfrac{\dfrac{\dfrac{\vdots}{u \xrightarrow{[v/a]} z}}{(\textbf{fn}\ a \Rightarrow u) \xrightarrow{\Box v} z}}{(\textbf{fn}\ a \Rightarrow u)\ v \longrightarrow z}\ (ap3).$$

By the induction hypothesis, there must be a proof that x' can do a $\Box$ transition therefore x' is of the form $(\textbf{fn}\ a' \Rightarrow u')$. Since v is a value we know that $v \xrightarrow{\text{v}} v$, therefore by the induction hypothesis $v' \xrightarrow{\text{v}} v'$. Therefore we have:

$$\dfrac{\dfrac{\dfrac{\vdots}{u' \xrightarrow{[v'/a]} z'}}{(\textbf{fn}\ a' \Rightarrow u') \xrightarrow{\Box v'} z'}}{(\textbf{fn}\ a' \Rightarrow u')\ v' \longrightarrow z'}\ (ap3).$$

By proposition 2.1, $z = u[v/a]$ and $z' = u'[v'/a']$. By the definition of R', $u[v/a]\ R'\ u[v'/a]$. Since $(\textbf{fn}\ a \Rightarrow u)\ R_i\ (\textbf{fn}\ a' \Rightarrow u')$, and $(\textbf{fn}\ a \Rightarrow u) \xrightarrow{\Box v'} u[v'/a]$ and $(\textbf{fn}\ a' \Rightarrow u') \xrightarrow{\Box v'} u'[v'/a']$ it follows from the induction hypothesis that $u[v'/a]\ R'\ u'[v'/a']$. Therefore by the transitive property of R', $u[v/a]\ R'\ u'[v'/a']$ that is $z\ R'\ z'$.

Case 4: $w =$ if x then y else z and $w' =$ if x' then y' else z' where $x\ R_i\ x'$, $y\ R_i\ y'$ and $z\ R_i\ z'$ For any provable transition of if x then y else z, the last rule in the proof is (sub7), (if1), (if2) or (if3).

Case 4a: (sub7) If last rule is (sub7) then by proposition 2.1 $z = w[e/a]$ and $z' = w'[e/a]$ and by the previous lemma $z\ R_{i+1}\ z'$.

Case 4b: (if1) If the last rule in the proof is (if1) then the proof is of the form:

$$\dfrac{\dfrac{\vdots}{x \longrightarrow u}}{\text{if } x \text{ then } y \text{ else } z \longrightarrow \text{if } u \text{ then } y \text{ else } z}\ (if1).$$

By the induction hypothesis, $x' \longrightarrow u'$ such that $u\ R'\ u'$. So we have:

$$\dfrac{\dfrac{\vdots}{x' \longrightarrow u'}}{\text{if } x' \text{ then } y' \text{ else } z' \longrightarrow \text{if } u' \text{ then } y' \text{ else } z'}\ (\mathit{if1}).$$

where by case 6 of the definition of R', if u then y else z R' if u' then y' else z'.

Case 4c: (if2) If the last rule in the proof is (if2) then the proof is of the form:

$$\dfrac{x \xrightarrow{k} x \quad k \neq 0}{\text{if } x \text{ then } y \text{ else } z \longrightarrow y}\ (\mathit{if2}).$$

By the induction hypothesis, $x' \xrightarrow{k} x'$. So we have:

$$\dfrac{x' \xrightarrow{k} x' \quad k \neq 0}{\text{if } x' \text{ then } y' \text{ else } z' \longrightarrow y}\ (\mathit{if2}),$$

where by case 3 of the definition of R', $y\ R'\ y'$.

Case 4d: (if3) If the last rule in the proof is (if3) then the proof is of the form:

$$\dfrac{x \xrightarrow{0} x}{\text{if } x \text{ then } y \text{ else } z \longrightarrow z}\ (\mathit{if3}).$$

By the induction hypothesis, $x' \xrightarrow{0} x'$. So we have:

$$\dfrac{x' \xrightarrow{0} x'}{\text{if } x' \text{ then } y' \text{ else } z' \longrightarrow z'}\ (\mathit{if3}).$$

where by case 3 of the definition of R', $z\ R'\ z'$.

Case 5: $w = x[y/b]$ and $w' = x[y'/b]$ where $y\ R_i\ y'$
We prove will this case by structural induction on x.

Case 5 (Base): If x is a constant then $x[y/a] = x = x[y'/a]$ and the result is trivially true. If x is an identifier other than a then again $x[y/a] = x = x[y'/a]$ and the result is trivially true. If x is the identifier a, then $x[y/a] = y$ and $x[y'/a] = y'$ and since $y\ R_i\ y'$, by the induction hypothesis the result is true.

For all of the cases, if the transition is a substitution transition then by proposition 2.1 $z = w[e/a]$ and $z' = w'[e/a]$ and by the previous lemma $z\ R_{i+1}\ z'$.

Case 5a: $(x = (\textbf{fn}\ b => u))$
If x is the function definition $(\textbf{fn}\ b => u)$ then there are two possible last rules in a proof of a transition of x, (tp4) and (tp5).

If (tp4) is the last rule in the proof, then the proof is of the form:

$$(\mathbf{fn}\ b => u)[y/a] \xrightarrow{\text{v}} (\mathbf{fn}\ b => u)[y/a] \quad (tp4).$$

It follows immediately that:

$$(\mathbf{fn}\ b => u)[y'/a] \xrightarrow{\text{v}} (\mathbf{fn}\ b => u)[y'/a] \quad (tp4)$$

where from the definition of R', $(\mathbf{fn}\ b => u)[y/a]\ R'\ (\mathbf{fn}\ b => u)[y'/a]$.

If (tp5) is the last rule in the proof, then we need to consider two cases. If $a = b$ then $(\mathbf{fn}\ b => u)[y/a] = (\mathbf{fn}\ b => u)$ and $(\mathbf{fn}\ b => u)[y'/a] = (\mathbf{fn}\ b => u)$ and the result is immediate.

If $a \neq b$ then $(\mathbf{fn}\ b => u)[y/a] = (\mathbf{fn}\ c => u[c/b][y/a])$ and $(\mathbf{fn}\ b => u)[y'/a] = (\mathbf{fn}\ c' => u[c'/b][y'/a])$, where c is the first identifier that does not occur in $(\mathbf{fn}\ b => u)[y/a]$ or e and c' is the first identifier that does not occur in $(\mathbf{fn}\ b => u)[y'/a]$ or e. In this case the proof is of the form:

$$\frac{b \xrightarrow{b} b \quad \dfrac{\vdots}{u[c/b][y/a] \xrightarrow{[e/c]} u[c/b][y/a][e/c]}}{(\mathbf{fn}\ c => u[c/b][y/a]) \xrightarrow{\Box e} u[c/b][y/a][e/c]}\ (tp5).$$

By proposition 2.1, there is also a proof that:

$$\frac{b \xrightarrow{b} b \quad \dfrac{\vdots}{u[c'/b][y'/a] \xrightarrow{[e/c']} u[c'/b][y'/a][e/c']}}{(\mathbf{fn}\ c' => u[c'/b][y'/a]) \xrightarrow{\Box e} u[c'/b][y'/a][e/c']}\ (tp5),$$

Let d be an identifier that does not occur in $(\mathbf{fn}\ b => u)[y/a]$, $(\mathbf{fn}\ b => u)[y'/a]$ or e. Since d is an identifier, the nesting depth of function symbols is the same in u and $u[d/b]$. Therefore by the induction hypothesis, $u[d/b][y/a]\ R'\ u[d/b][y'/a]$ and by the previous lemma $u[d/b][y/a][e/d]\ R'\ u[d/b][y'/a][e/d]$. Where by the properties of substitution $u[d/b][y/a][e/d] = u[c/b][y/a][e/c]$ and $u[d/b][y'/a][e/d] = u[c'/b][y'/a][e/c']$.

Case 5b: $(x = u\ v)$

If x is the application, $(u\ v)$ then $x[y/a] = (u\ v)[y/a] = u[y/a]\ v[y/a]$ and $x[y'/a] = (u\ v)[y'/a] = u[y'/a]\ v[y'/a]$. By case 5 of the definition of R_{i+1}, we have $u[y/a]\ R_{i+1}\ u[y'/a]$ and $v[y/a]\ R_{i+1}\ v[y'/a]$. Again we have to consider each possible form of transition. The last rule in the proof can be (ap1), (ap2) or (ap3).

If (ap1) is the last rule in the proof of the transition , then the proof is of the form:

$$\frac{\dfrac{\vdots}{u[y/a] \longrightarrow w} \quad v[y/a] \xrightarrow{\text{v}} v[y/a]}{u[y/a]\ v[y/a] \longrightarrow w\ v[y/a]}\ (ap1)$$

By the induction hypothesis, $u[y'/a] \longrightarrow w'$ such that $w\ R'\ w'$ and $v[y'/a] \xrightarrow{\mathrm{v}} v[y'/a]$. Therefore we have:

$$\frac{\dfrac{\vdots}{u[y'/a] \longrightarrow w'} \quad v[y'/a] \xrightarrow{\mathrm{v}} v[y'/a]}{u[y'/a]\ v[y'/a] \longrightarrow w'\ v[y'/a]}\ (ap1)$$

where, by case 5 of the definition of R', $(w\ v[y/a])\ R'\ (w'\ v[y'/a])$.

If (ap2) is the last rule in the proof of the transition , then the proof is of the form:

$$\frac{\dfrac{\vdots}{v[y/a] \longrightarrow w}}{u[y/a]\ v[y/a] \longrightarrow u[y/a]\ w}\ (ap2).$$

By the induction hypothesis, $v[y'/a] \longrightarrow w'$ such that $w\ R'\ w'$. Therefore we have:

$$\frac{\dfrac{\vdots}{v[y'/a] \longrightarrow w'}}{u[y'/a]\ v[y'/a] \longrightarrow u[y'/a]\ w'}\ (ap2).$$

where, by case 5 of the definition of R', $(u[y/a]\ w)\ R'\ (u[y'/a]\ w')$.

If (ap3) is the last rule in the proof of the transition , then we know $u = (\mathbf{fn}\ b => w)[y/a]$. If $b = a$ then $(\mathbf{fn}\ a => w)[y/a] = (\mathbf{fn}\ a => w)$ and the proof is of the form:

$$\frac{w \xrightarrow{\square v[y/a]} w[v[y/a]/a]}{(\mathbf{fn}\ a => w)\ v[y/a] \longrightarrow w[v[y/a]/a]}\ (ap3).$$

Therefore we have:

$$\frac{w \xrightarrow{\square v[y'/a]} w[v[y'/a]/a]}{(\mathbf{fn}\ a => w)\ v[y'/a] \longrightarrow w[v[y'/a]/a]}\ (ap3),$$

where by case 7 of the definition of R', $v[y/a]\ R'\ v[y'/a]$ and again by case 7 of the definition of R' $w[v[y/a]/a]\ R'\ w[v[y'/a]/a]$.

If $b \neq a$ then $(\mathbf{fn}\ b => w)[y/a] = (\mathbf{fn}\ c => w[c/b][y/a])$ and the proof is of the form:

$$\frac{w[c/b][y/a] \xrightarrow{\square v[y/a]} w[c/b][y/a][v[y/a]/c]}{(\mathbf{fn}\ c => w[c/b][y/a])\ v[y/a] \longrightarrow w[c/b][y/a][v[y/a]/c]}\ (ap3).$$

We know that by reversing the order of substitutions, we have

$$w[c/b][y/a][v[y/a]/c]\ S\ w[c/b][v/c][y/a]\ S\ w[v/b][y/a].$$

Therefore we have:

$$\frac{(\textbf{fn } c' => w[c'/b][y'/a]) \stackrel{\Box v[y'/a]}{\longrightarrow} w[c'/b][y'/a][v[y'/a]/c']}{(\textbf{fn } c' => w[c'/b][y'/a])\ v[y'/a] \longrightarrow w[c'/b][y'/a][v[y'/a]/c']}\ (\mathit{ap3}),$$

where $w[c'/b][y'/a][v[y'/a]/c']\ S\ w[c'/b][v/c'][y'/a]\ S\ w[v/b][y'/a]$ and by case 7 of the definition of R' we have $w[v/b][y/a]\ R'\ w[v/b][y'/a]$.

Case 5c: ($x =$ if u then v else w)
If x is the conditional, (if u then v else w) then

$$x[y/a] = (\text{if } u \text{ then } v \text{ else } w)[y/a] = (\text{if } u[y/a] \text{ then } v[y/a] \text{ else } w[y/a])$$

and

$$x[y'/a] = (\text{if } u \text{ then } v \text{ else } w)[y'/a] = (\text{if } u[y'/a] \text{ then } v[y'/a] \text{ else } w[y'/a]).$$

By the definition of R_{i+1}, $u[y/a]\ R_{i+1}\ u[y'/a]$, $v[y/a]\ R_{i+1}\ v[y'/a]$ and $w[y/a]\ R_{i+1}\ w[y'/a]$. The last rule in the proof can be (if1), (if2) or (if3).

If the last rule in the proof is (if1) then the proof is of the form:

$$\frac{\dfrac{\vdots}{u[y/a] \longrightarrow t}}{\text{if } u[y/a] \text{ then } v[y/a] \text{ else } w[y/a] \longrightarrow \text{if } t \text{ then } v[y/a] \text{ else } w[y/a]}\ (\mathit{if1}).$$

By the induction hypothesis, there is a proof of $u[y'/a] \longrightarrow t'$ such that $t\ R'\ t'$. Therefore we have:

$$\frac{\dfrac{\vdots}{u[y'/a] \longrightarrow t'}}{\text{if } u[y'/a] \text{ then } v[y'/a] \text{ else } w[y'/a] \longrightarrow \text{if } t' \text{ then } v[y'/a] \text{ else } w[y'/a]}\ (\mathit{if1}),$$

where by the definition of R', (if t then $v[y/a]$ else $w[y/a]$ R' if t' then $v[y'/a]$ else $w[y'/a]$).

If the last rule in the proof is (if2) then the proof is of the form:

$$\frac{u[y/a] \stackrel{k}{\longrightarrow} u[y/a] \quad k \neq 0}{\text{if } u[y/a] \text{ then } v[y/a] \text{ else } w[y/a] \longrightarrow v[y/a]}\ (\mathit{if2}).$$

By the induction hypothesis, there is a proof of $u[y'/a] \stackrel{k}{\longrightarrow} u[y'/a]$. Therefore we have:

$$\frac{u[y'/a] \stackrel{k}{\longrightarrow} u[y'/a] \quad k \neq 0}{\text{if } u[y'/a] \text{ then } v[y'/a] \text{ else } w[y'/a] \longrightarrow v[y'/a]}\ (\mathit{if2}).$$

where by the definition of R', $v[y/a]\ R'\ v[y'/a]$.

If the last rule in the proof is (if3) then the proof is of the form:

$$\frac{u[y/a] \xrightarrow{0} u[y/a]}{\text{if } u[y/a] \text{ then } v[y/a] \text{ else } w[y/a] \longrightarrow w[y/a]} \ (\mathit{if2}).$$

By the induction hypothesis, there is a proof of $u[y'/a] \xrightarrow{k} u[y'/a]$. Therefore we have:

$$\frac{u[y'/a] \xrightarrow{k} u[y'/a]}{\text{if } u[y'/a] \text{ then } v[y'/a] \text{ else } w[y'/a] \longrightarrow w[y'/a]} \ (\mathit{if2}).$$

where by the definition of R', $w[y/a]\ R'\ w[y'/a]$.
□

E Proposition 3.2

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

Proof: Any proof of a transition of $\kappa\langle\!\langle x\rangle\!\rangle$, will have an evaluation rule or a typing rule as the last step, since those are the only rules with the debugging state function symbol on the left-hand side of the conclusion. We will do the proof by case analysis on the last rule in the proof of the transition.

Case 1: If the last step in the proof is by (db1) then we have:

$$\frac{\dfrac{\phi}{\kappa \Longrightarrow \kappa'}}{\kappa\langle\!\langle x\rangle\!\rangle \Longrightarrow \kappa'\langle\!\langle x\rangle\!\rangle} \ (\mathit{db1}) \qquad\qquad \frac{\dfrac{\phi}{\kappa \Longrightarrow \kappa'}}{\kappa\langle\!\langle x'\rangle\!\rangle \Longrightarrow \kappa'\langle\!\langle x'\rangle\!\rangle} \ (\mathit{db1})$$

where $\kappa'\langle\!\langle x\rangle\!\rangle \sim \kappa'\langle\!\langle x'\rangle\!\rangle$.

Case 2: If the last step in the proof is by (db2) then we have:

$$\frac{\dfrac{\phi}{\kappa \overset{!}{\Longrightarrow} \kappa'} \quad \dfrac{\vdots}{x \longrightarrow y}}{\kappa\langle\!\langle x\rangle\!\rangle \Longrightarrow \kappa\langle\!\langle y\rangle\!\rangle} \ (\mathit{db2}) \qquad\qquad \frac{\dfrac{\phi}{\kappa \overset{!}{\Longrightarrow} \kappa'} \quad \dfrac{\vdots}{x' \longrightarrow y'}}{\kappa\langle\!\langle x'\rangle\!\rangle \Longrightarrow \kappa\langle\!\langle y'\rangle\!\rangle} \ (\mathit{db2})$$

Since $x \sim x'$, if there is a proof that $x \longrightarrow y$ then there is a proof that $x' \longrightarrow y'$ such that $y \sim y'$. Likewise if there is a proof that $x' \longrightarrow y'$ then there is a proof that $x \longrightarrow y$ such that $y \sim y'$. Therefore $\kappa\langle\!\langle y\rangle\!\rangle \sim \kappa\langle\!\langle y'\rangle\!\rangle$.

Case 3: If the last step in the proof is by (db3) then we have:

$$\dfrac{\dfrac{\phi}{\kappa \xRightarrow{[e/a]} \kappa'} \quad \dfrac{\vdots}{x \xrightarrow{[e/a]} y}}{\kappa\llparenthesis x \rrparenthesis \Longrightarrow \kappa'\llparenthesis y \rrparenthesis}\ (db3) \qquad \dfrac{\dfrac{\phi}{\kappa \xRightarrow{[e/a]} \kappa'} \quad \dfrac{\vdots}{x' \xrightarrow{[e/a]} y'}}{\kappa\llparenthesis x' \rrparenthesis \Longrightarrow \kappa'\llparenthesis y' \rrparenthesis}\ (db3)$$

Since $x \sim x'$, if there is a proof that $x \xrightarrow{[e/a]} y$ then there is a proof that $x' \xrightarrow{[e/a]} y'$ such that $y \sim y'$. Likewise if there is a proof that $x' \xrightarrow{[e/a]} y'$ then there is a proof that $x \xrightarrow{[e/a]} y$ such that $y \sim y'$. Therefore $\kappa'\llparenthesis y \rrparenthesis\sim \kappa'\llparenthesis y' \rrparenthesis$.

Case 4: If the last step in the proof is by (db4) then we have:

$$\dfrac{\dfrac{\phi}{\kappa \xRightarrow{!} \kappa'} \quad \dfrac{\vdots}{x \xrightarrow{\Box v} y}}{\kappa{:}\{\text{ - }\ v\}\llparenthesis x \rrparenthesis \Longrightarrow \kappa\llparenthesis y \rrparenthesis}\ (db4) \qquad \dfrac{\dfrac{\phi}{\kappa \xRightarrow{!} \kappa'} \quad \dfrac{\vdots}{x' \xrightarrow{\Box v} y'}}{\kappa{:}\{\text{ - }\ v\}\llparenthesis x' \rrparenthesis \Longrightarrow \kappa\llparenthesis y' \rrparenthesis}\ (db4)$$

Since $x \sim x'$, if there is a proof that $x \xrightarrow{\Box v} y$ then there is a proof that $x' \xrightarrow{\Box v} y'$ such that $y \sim y'$. Likewise if there is a proof that $x' \xrightarrow{\Box v} y'$ then there is a proof that $x \xrightarrow{\Box v} y$ such that $y \sim y'$. Therefore $\kappa\llparenthesis y \rrparenthesis\sim \kappa\llparenthesis y' \rrparenthesis$.

Case 5: If the last step in the proof is by (db5) then we have:

$$\dfrac{\dfrac{\phi_1}{\kappa \xRightarrow{!} \kappa'} \quad \dfrac{\phi_2}{u \longrightarrow w} \quad \dfrac{\vdots}{x \xrightarrow{\mathrm{v}} x}}{\kappa{:}\{u\ \text{ - }\}\llparenthesis x \rrparenthesis \Longrightarrow \kappa{:}\{w\ \text{ - }\}\llparenthesis x \rrparenthesis}\ (db5) \qquad \dfrac{\dfrac{\phi_1}{\kappa \xRightarrow{!} \kappa'} \quad \dfrac{\phi_2}{u \longrightarrow w} \quad \dfrac{\vdots}{x' \xrightarrow{\mathrm{v}} x'}}{\kappa{:}\{u\ \text{ - }\}\llparenthesis x' \rrparenthesis \Longrightarrow \kappa{:}\{w\ \text{ - }\}\llparenthesis x' \rrparenthesis}\ (db5)$$

Since $x \sim x'$, if there is a proof that $x \xrightarrow{\mathrm{v}} x$ then there is a proof that $x' \xrightarrow{\mathrm{v}} x'$. Likewise if there is a proof that $x' \xrightarrow{\mathrm{v}} x'$ then there is a proof that $x \xrightarrow{\mathrm{v}} x$. Therefore $\kappa{:}\{w\ \text{ - }\}\llparenthesis x \rrparenthesis\sim \kappa{:}\{w\ \text{ - }\}\llparenthesis x' \rrparenthesis$.

Case 6: If the last step in the proof is by (db6) then we have:

$$\dfrac{\dfrac{\phi_1}{\kappa \xRightarrow{!} \kappa'} \quad \dfrac{\phi_2}{u \xrightarrow{\Box x} y}}{\kappa{:}\{u\ \text{ - }\}\llparenthesis x \rrparenthesis \Longrightarrow \kappa\llparenthesis y \rrparenthesis}\ (db6).$$

Therefore u must be a function definition and both of the following proofs exist:

$$\dfrac{\dfrac{\phi_2}{u \xrightarrow{\Box x} y}}{u\ x \longrightarrow y}\ (ap3) \qquad \dfrac{\dfrac{\phi_3}{u \xrightarrow{\Box x'} y'}}{u\ x' \longrightarrow y'}\ (ap3).$$

Since $x \sim x'$ and by proposition 2.8 bisimulation is a congruence, it follows that $y \sim y'$. Therefore:

$$\frac{\dfrac{\phi_1}{\kappa \stackrel{!}{\Longrightarrow} \kappa'} \quad \dfrac{\phi_3}{u \xrightarrow{\Box x'} y'}}{\kappa\colon \{u\ \text{-}\ \}\langle\!|x'|\!\rangle \Longrightarrow \kappa\langle\!|y'|\!\rangle}\ (db6),$$

where $\kappa\langle\!|y|\!\rangle \sim \kappa\langle\!|y'|\!\rangle$. Likewise, the proof in the other direction holds.

Case 7: If the last step in the proof is by (db7) then we have:

$$\frac{\dfrac{\phi}{\kappa \stackrel{!}{\Longrightarrow} \kappa'} \quad \dfrac{\vdots}{x \xrightarrow{k} x} \quad k \neq 0}{\kappa\colon \{\ \text{-}\ ?y, z\}\langle\!|x|\!\rangle \Longrightarrow \kappa\langle\!|y|\!\rangle}\ (db7) \qquad \frac{\dfrac{\phi}{\kappa \stackrel{!}{\Longrightarrow} \kappa'} \quad \dfrac{\vdots}{x' \xrightarrow{k} x'} \quad k \neq 0}{\kappa\colon \{\ \text{-}\ ?y, z\}\langle\!|x'|\!\rangle \Longrightarrow \kappa\langle\!|y|\!\rangle}\ (db7)$$

Since $x \sim x'$, if there is a proof that $x \xrightarrow{\mathrm{k}} x$ then there is a proof that $x' \xrightarrow{\mathrm{k}} x'$. Likewise if there is a proof that $x' \xrightarrow{\mathrm{k}} x'$ then there is a proof that $x \xrightarrow{\mathrm{k}} x$. Therefore $\kappa\langle\!|y|\!\rangle \sim \kappa\langle\!|y|\!\rangle$.

Case 8: If the last step in the proof is by (db8) then the proof is of the form:

$$\frac{\dfrac{\phi}{\kappa \stackrel{!}{\Longrightarrow} \kappa'} \quad \dfrac{\vdots}{x \xrightarrow{0} x}}{\kappa\colon \{\ \text{-}\ ?y, z\}\langle\!|x|\!\rangle \Longrightarrow \kappa\langle\!|z|\!\rangle}\ (db8) \qquad \frac{\dfrac{\phi}{\kappa \stackrel{!}{\Longrightarrow} \kappa'} \quad \dfrac{\vdots}{x' \xrightarrow{0} x'}}{\kappa\colon \{\ \text{-}\ ?y, z\}\langle\!|x'|\!\rangle \Longrightarrow \kappa\langle\!|z|\!\rangle}\ (db8)$$

Since $x \sim x'$, if there is a proof that $x \xrightarrow{0} x$ then there is a proof that $x' \xrightarrow{0} x'$. Likewise if there is a proof that $x' \xrightarrow{0} x'$ then there is a proof that $x \xrightarrow{0} x$. Therefore $\kappa\langle\!|y|\!\rangle \sim \kappa\langle\!|y|\!\rangle$.

Case 9: If the last step in the proof is by (db9) then the proof is of the form:

$$\frac{\dfrac{\phi}{\kappa \stackrel{*}{\Longrightarrow} d}}{\kappa\langle\!|x|\!\rangle \Longrightarrow d}\ (db9) \qquad \frac{\dfrac{\phi}{\kappa \stackrel{*}{\Longrightarrow} d}}{\kappa\langle\!|x'|\!\rangle \Longrightarrow d}\ (db9)$$

where $d \sim d$.

Case 10: If the last step in the proof is by (dt1) then we have:

$$\frac{\dfrac{\vdots}{x \xrightarrow{k} x}}{\epsilon\langle\!|x|\!\rangle \stackrel{k}{\Longrightarrow} \epsilon\langle\!|x|\!\rangle}\ (dt1) \qquad \frac{\dfrac{\vdots}{x' \xrightarrow{k} x'}}{\epsilon\langle\!|x'|\!\rangle \stackrel{k}{\Longrightarrow} \epsilon\langle\!|x'|\!\rangle}\ (dt1)$$

Since $x \sim x'$, if there is a proof that $x \xrightarrow{k} x$ then there is a proof that $x' \xrightarrow{k} x'$. Likewise if there is a proof that $x' \xrightarrow{k} x'$ then there is a proof that $x \xrightarrow{k} x$. Therefore $\kappa\langle\!|x|\!\rangle \sim \kappa\langle\!|x'|\!\rangle$.

Case 11: If the last step in the proof is by (dt2) then we have:

$$\frac{x \xrightarrow{a} x}{\epsilon\langle\!|x|\!\rangle \stackrel{a}{\Longrightarrow} \epsilon\langle\!|x|\!\rangle}\ (dt2). \qquad \frac{x' \xrightarrow{a} x'}{\epsilon\langle\!|x'|\!\rangle \stackrel{a}{\Longrightarrow} \epsilon\langle\!|x'|\!\rangle}\ (dt2),$$

Since $x \sim x'$, if there is a proof that $x \xrightarrow{a} x$ then there is a proof that $x' \xrightarrow{a} x'$. Likewise if there is a proof that $x' \xrightarrow{a} x'$ then there is a proof that $x \xrightarrow{a} x$. Therefore $\kappa\langle\!|x|\!\rangle\sim \kappa\langle\!|x'|\!\rangle$.

Case 12: If the last step in the proof is by (dt3) then we have:

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

Since $x \sim x'$, if there is a proof that $x \xrightarrow{\mathrm{v}} x$ then there is a proof that $x' \xrightarrow{\mathrm{v}} x'$. Likewise if there is a proof that $x' \xrightarrow{\mathrm{v}} x'$ then there is a proof that $x \xrightarrow{\mathrm{v}} x$. Therefore $\kappa\langle\!|x|\!\rangle\sim \kappa\langle\!|x'|\!\rangle$.

Case 13: If the last step in the proof is by (dt4) then we have:

$$\frac{\dfrac{\vdots}{x \xrightarrow{\Box e} y}}{\epsilon\langle\!|x|\!\rangle \stackrel{\Box e}{\Longrightarrow} \epsilon\langle\!|y|\!\rangle}\ (dt4) \qquad \frac{\dfrac{\vdots}{x' \xrightarrow{\Box e} y'}}{\epsilon\langle\!|x'|\!\rangle \stackrel{\Box e}{\Longrightarrow} \epsilon\langle\!|y'|\!\rangle}\ (dt4)$$

Since $x \sim x'$, if there is a proof that $x \xrightarrow{\Box e} x$ then there is a proof that $x' \xrightarrow{\Box e} x'$. Likewise if there is a proof that $x' \xrightarrow{\Box e} x'$ then there is a proof that $x \xrightarrow{\Box e} x$. Therefore $\kappa\langle\!|x|\!\rangle\sim \kappa\langle\!|x'|\!\rangle$.
$\Box$

F Theorem 3.1

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

Proof: Let S relate debugging terms $\kappa\langle\!|x|\!\rangle$ and $\kappa\langle\!|x'|\!\rangle$ iff x and x' are identical up to the renaming of bound identifiers. Let I be the identity relation. Let R be the transitive closure of $S \cup I \cup \{(d, d'){:}\, d \stackrel{\delta}{\Downarrow} d'\}$. We will show that R' is a bisimulation relation. Since S and I are bisimulation relations, it is sufficient to show $\{(d, d') : d \stackrel{\delta}{\Downarrow} d'\}$ is a bisimulation relation. By inspection of the focusing rules, we know that the programming language expression in focus must be an application, function definition or conditional expression. Since the only such expression that can do a typing rule is a function definition and from the focusing rule we know in this case the debugging context is not empty, therefore the last rule in a proof of a transition of d, cannot be a typing rule and must in fact be an evaluation rule. Similarly, the debugging context for d' cannot be empty, therefore the last rule in a proof of a transition of

d', cannot be a typing rule and must in fact be an evaluation rule. We will do the proof by a case analysis on the last rule in the proof of the possible transitions. First we will consider the possible transitions of d.

Case 1:(db1)

First we will consider if $\delta \in \{\ l, r, \text{if, then or else}\ \}$ then we will consider the case where $\delta = \lambda$.

Case 1a: $d = \kappa(\!|x|\!)$, $d' = \kappa\colon c(\!|x'|\!)$, $\delta \in \{\ l, r, \text{if, then or else}\ \}$

In this case the proof is of the form:

$$\dfrac{\dfrac{\phi}{\kappa \Longrightarrow \kappa'}}{\kappa(\!|x|\!) \Longrightarrow \kappa'(\!|x|\!)}\ (db1) \qquad\qquad \dfrac{\dfrac{\dfrac{\phi}{\kappa \Longrightarrow \kappa'}}{\kappa\colon c \Longrightarrow \kappa'\colon c}\ (ke1)}{\kappa\colon c(\!|x'|\!) \Longrightarrow \kappa'\colon c(\!|x'|\!)}\ (db1),$$

where $\kappa'(\!|x|\!) \overset{\delta}{\Downarrow} \kappa'\colon c(\!|x'|\!)$.

Case 1b: $d = \kappa\colon\{\text{ - }\ y\}(\!|(\textbf{fn}\ a => x)|\!)$, $d' = \kappa\colon\{y/a\}(\!|x|\!)$, $\delta = \lambda$

The proof can take one of two forms since the second to last rule in the proof can be either (ke1) or (ke2). If the second to last rule is (ke1), then the proof is of the form:

$$\dfrac{\dfrac{\dfrac{\phi}{\kappa \Longrightarrow \kappa'}}{\kappa\colon\{\text{ - }\ y\} \Longrightarrow \kappa'\colon\{\text{ - }\ y\}}\ (ke1)}{\kappa\colon\{\text{ - }\ y\}(\!|(\textbf{fn}\ a => x)|\!) \Longrightarrow \kappa'\colon\{\text{ - }\ y\}(\!|(\textbf{fn}\ a => x)|\!)}\ (db1).$$

Then we have:

$$\dfrac{\dfrac{\dfrac{\phi}{\kappa \Longrightarrow \kappa'}}{\kappa\colon\{y/a'\} \Longrightarrow \kappa'\colon\{y/a'\}}\ (ke1)}{\kappa\colon\{y/a'\}(\!|x[a'/a]|\!) \Longrightarrow \kappa'\colon\{y/a'\}(\!|x[a'/a]|\!)}\ (db1),$$

where $\kappa'\colon\{\text{ - }\ y\}(\!|(\textbf{fn}\ a => x)|\!) \overset{\lambda}{\Downarrow} \kappa'\colon\{y/a'\}(\!|x[a'/a]|\!)$.

If the second to last rule is (ke2), then the proof is of the form:

$$\dfrac{\dfrac{\dfrac{\phi_1}{\kappa \overset{!}{\Longrightarrow} \kappa} \quad \dfrac{\phi_2}{y \longrightarrow y'}}{\kappa\colon\{\text{ - }\ y\} \Longrightarrow \kappa\colon\{\text{ - }\ y'\}}\ (ke2)}{\kappa\colon\{\text{ - }\ y\}(\!|(\textbf{fn}\ a => x)|\!) \Longrightarrow \kappa\colon\{\text{ - }\ y'\}(\!|(\textbf{fn}\ a => x)|\!)}\ (db1).$$

Then we have:

$$\dfrac{\dfrac{\dfrac{\phi_1}{\kappa \stackrel{!}{\Longrightarrow} \kappa} \quad \dfrac{\phi_2}{y \longrightarrow y'}}{\kappa\colon\{y/a'\} \Longrightarrow \kappa\colon\{y'/a'\}}\ (ke3)}{\kappa\colon\{y/a'\}\llparenthesis x[a'/a]\rrparenthesis \Longrightarrow \kappa\colon\{y'/a'\}\llparenthesis x[a'/a]\rrparenthesis}\ (db1),$$

where $\kappa\colon\{\text{-}\ \ y'\}\llparenthesis(\mathbf{fn}\ a \Rightarrow x)\rrparenthesis \stackrel{\lambda}{\Downarrow} \kappa\colon\{y'/a'\}\llparenthesis x[a'/a]\rrparenthesis$.

Case 2: (db2)

The second to last rule in the proof of the transition of d can be any of the six program language evaluation rules.

Case 2a: (ap1)

In this case, $d = \kappa\llparenthesis x\ y\rrparenthesis$ and either $d' = \kappa\colon\{\text{-}\ \ y\}\llparenthesis x\rrparenthesis$ or $\kappa\colon\{x\ \ \text{-}\}\llparenthesis y\rrparenthesis$ depending on whether $\delta = l$ or $\delta = r$. The proof has the form:

$$\dfrac{\dfrac{\phi_1}{\kappa \stackrel{!}{\Longrightarrow} \kappa'} \quad \dfrac{\dfrac{\phi_2}{x \longrightarrow x'} \quad y \stackrel{v}{\longrightarrow} y}{x\ y \longrightarrow x'\ y}\ (ap1)}{\kappa\llparenthesis x\ y\rrparenthesis \Longrightarrow \kappa\llparenthesis x'\ y\rrparenthesis}\ (db2).$$

In the first case:

$$\dfrac{\dfrac{\dfrac{\phi_1}{\kappa \stackrel{!}{\Longrightarrow} \kappa'} \quad y \stackrel{v}{\longrightarrow} y'}{\kappa\colon\{\text{-}\ \ y\} \stackrel{!}{\Longrightarrow} \kappa\colon\{\text{-}\ \ y\}}\ (tr1) \quad \dfrac{\phi_2}{x \longrightarrow x'}}{\kappa\colon\{\text{-}\ \ y\}\llparenthesis x\rrparenthesis \Longrightarrow \kappa\colon\{\text{-}\ \ y\}\llparenthesis x'\rrparenthesis}\ (db2),$$

where $\kappa\llparenthesis x'\ y\rrparenthesis \stackrel{l}{\Downarrow} \kappa\colon\{\text{-}\ \ y\}\llparenthesis x'\rrparenthesis$. In the second case:

$$\dfrac{\dfrac{\phi_1}{\kappa \stackrel{!}{\Longrightarrow} \kappa} \quad \dfrac{\phi_2}{x \longrightarrow x'} \quad y \stackrel{v}{\longrightarrow} y}{\kappa\colon\{x\ \ \text{-}\}\llparenthesis y\rrparenthesis \Longrightarrow \kappa\colon\{x'\ \ \text{-}\}\llparenthesis y\rrparenthesis}\ (db5)$$

where $\kappa\llparenthesis x'\ y\rrparenthesis \stackrel{r}{\Downarrow} \kappa\colon\{x'\ \ \text{-}\}\llparenthesis y\rrparenthesis$.

Case 2b: (ap2)

In this case, $d = \kappa\llparenthesis x\ y\rrparenthesis$ and either $d' = \kappa\colon\{\text{-}\ \ y\}\llparenthesis x\rrparenthesis$ or $\kappa\colon\{x\ \ \text{-}\}\llparenthesis y\rrparenthesis$ depending on whether $\delta = l$ or $\delta = r$. The proof has the form:

$$\dfrac{\dfrac{\phi_1}{\kappa \stackrel{!}{\Longrightarrow} \kappa'} \quad \dfrac{\dfrac{\phi_2}{y \longrightarrow y'}}{x\ y \longrightarrow x\ y'}\ (ap2)}{\kappa\llparenthesis x\ y\rrparenthesis \Longrightarrow \kappa\llparenthesis x\ y'\rrparenthesis}\ (db2).$$

In the first case:

$$\cfrac{\cfrac{\cfrac{\phi_1}{\kappa \overset{!}{\Longrightarrow} \kappa'} \quad \cfrac{\phi_2}{y \longrightarrow y'}}{\kappa{:}\{\,\text{-}\ \ y\} \Longrightarrow \kappa{:}\{\,\text{-}\ \ y'\}}\,(ke2)}{\kappa{:}\{\,\text{-}\ \ y\}\langle\!| x |\!\rangle \Longrightarrow \kappa{:}\{\,\text{-}\ \ y'\}\langle\!| x |\!\rangle}\,(db1)$$

where $\kappa\langle\!| x\ y' |\!\rangle \overset{l}{\Downarrow} \kappa{:}\{\,\text{-}\ \ y'\}\langle\!| x |\!\rangle$. In the second case:

$$\cfrac{\cfrac{\cfrac{\phi_1}{\kappa \overset{!}{\Longrightarrow} \kappa}}{\kappa{:}\{x\ \ \text{-}\,\} \overset{!}{\Longrightarrow} \kappa{:}\{x\ \ \text{-}\,\}}\,(tr2) \quad \cfrac{\phi_2}{y \longrightarrow y'}}{\kappa{:}\{x\ \ \text{-}\,\}\langle\!| y |\!\rangle \Longrightarrow \kappa{:}\{x\ \ \text{-}\,\}\langle\!| y' |\!\rangle}\,(db2),$$

where $\kappa\langle\!| x\ y' |\!\rangle \overset{r}{\Downarrow} \kappa{:}\{x\ \ \text{-}\,\}\langle\!| y' |\!\rangle$.

Case 2c: (ap3)

In this case, $d = \kappa\langle\!| x\ y |\!\rangle$ and either $d' = \kappa{:}\{\,\text{-}\ \ y\}\langle\!| x |\!\rangle$ or $\kappa{:}\{x\ \ \text{-}\,\}\langle\!| y |\!\rangle$ depending on whether $\delta = l$ or $\delta = r$. The proof has the form:

$$\cfrac{\cfrac{\phi_1}{\kappa \overset{!}{\Longrightarrow} \kappa'} \quad \cfrac{\cfrac{\phi_2}{x \overset{\Box v}{\longrightarrow} x'}}{x\ v \longrightarrow x'}\,(ap3)}{\kappa\langle\!| x\ v |\!\rangle \Longrightarrow \kappa\langle\!| x' |\!\rangle}\,(db2).$$

In the first case:

$$\cfrac{\cfrac{\phi_1}{\kappa \overset{!}{\Longrightarrow} \kappa'} \quad \cfrac{\phi_1}{x \overset{\Box v}{\longrightarrow} x'}}{\kappa{:}\{\,\text{-}\ \ v\}\langle\!| x |\!\rangle \Longrightarrow \kappa\langle\!| x' |\!\rangle}\,(db4)$$

where $\kappa\langle\!| x' |\!\rangle = \kappa\langle\!| x' |\!\rangle$. In the second case:

$$\cfrac{\cfrac{\phi_1}{\kappa \overset{!}{\Longrightarrow} \kappa} \quad \cfrac{\phi_2}{x \overset{\Box v}{\longrightarrow} x'}}{\kappa{:}\{x\ \ \text{-}\,\}\langle\!| v |\!\rangle \Longrightarrow \kappa\langle\!| x' |\!\rangle}\,(db6)$$

where $\kappa\langle\!| x' |\!\rangle = \kappa\langle\!| x' |\!\rangle$.

Case 2d: (if1)

In this case, $d = \kappa\langle\!| \text{if } x \text{ then } y \text{ else } z |\!\rangle$ and d' can be any one of $\kappa{:}\{\,\text{-}\,?y, z\}\langle\!| x |\!\rangle$, $\kappa{:}\{x?\,\text{-}\,, z\}\langle\!| y |\!\rangle$ or $\kappa{:}\{x?y,\ \text{-}\,\}\langle\!| z |\!\rangle$. The proof has the form:

$$\cfrac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad \cfrac{x \longrightarrow x'}{\text{if } x \text{ then } y \text{ else } z \longrightarrow \text{if } x' \text{ then } y \text{ else } z}\,(if1)}{\kappa\langle\!| \text{if } x \text{ then } y \text{ else } z |\!\rangle \Longrightarrow \kappa\langle\!| \text{if } x' \text{ then } y \text{ else } z |\!\rangle}\,(db2)$$

In the first case:

$$\frac{\dfrac{\kappa \stackrel{!}{\Longrightarrow} \kappa'}{\kappa{:}\{\,\text{-}\,?y,z\} \stackrel{!}{\Longrightarrow} \kappa'{:}\{\,\text{-}\,?y,z\}}\,(tr3) \qquad x \longrightarrow x'}{\kappa{:}\{\,\text{-}\,?y,z\}\langle x\rangle \Longrightarrow \kappa{:}\{\,\text{-}\,?y,z\}\langle x'\rangle}\,(db2),$$

where $\kappa\langle \text{if } x' \text{ then } y \text{ else } z\rangle \stackrel{\text{if}}{\Downarrow} \kappa{:}\{\,\text{-}\,?y,z\}\langle x'\rangle$. In the second case:

$$\frac{\dfrac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad x \longrightarrow x'}{\kappa{:}\{x?\,\text{-}\,,z\} \Longrightarrow \kappa{:}\{x'?\,\text{-}\,,z\}}\,(ke4)}{\kappa{:}\{x?\,\text{-}\,,z\}\langle y\rangle \Longrightarrow \kappa{:}\{x'?\,\text{-}\,,z\}\langle y\rangle}\,(db1)$$

where $\kappa\langle \text{if } x' \text{ then } y \text{ else } z\rangle \stackrel{\text{then}}{\Downarrow} \kappa{:}\{x'?\,\text{-}\,,z\}\langle y\rangle$. In the third case:

$$\frac{\dfrac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad x \longrightarrow x'}{\kappa{:}\{x?y,\,\text{-}\,\} \Longrightarrow \kappa{:}\{x'?y,\,\text{-}\,\}}\,(ke6)}{\kappa{:}\{x?y,\,\text{-}\,\}\langle y\rangle \Longrightarrow \kappa{:}\{x'?y,\,\text{-}\,\}\langle z\rangle}\,(db1)$$

where $\kappa\langle \text{if } x' \text{ then } y \text{ else } z\rangle \stackrel{\text{else}}{\Downarrow} \kappa{:}\{x'?\,\text{-}\,,y\}\langle z\rangle$.

Case 2e: (if2)

In this case, $d = \kappa\langle \text{if } x \text{ then } y \text{ else } z\rangle$ and d' can be any one of $\kappa{:}\{\,\text{-}\,?y,z\}\langle x\rangle$, $\kappa{:}\{x?\,\text{-}\,,z\}\langle y\rangle$ or $\kappa{:}\{x?y,\,\text{-}\,\}\langle z\rangle$. The proof has the form:

$$\frac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad \dfrac{x \stackrel{k}{\longrightarrow} x' \quad k \neq 0}{\text{if } x \text{ then } y \text{ else } z \longrightarrow y}\,(if2)}{\kappa\langle \text{if } x \text{ then } y \text{ else } z\rangle \Longrightarrow \kappa\langle y\rangle}\,(db2)$$

In the first case:

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

where $\kappa\langle y\rangle = \kappa\langle y\rangle$. In the second case:

$$\frac{\dfrac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad x \stackrel{k}{\longrightarrow} x' \quad k \neq 0}{\kappa{:}\{x?\,\text{-}\,,z\} \Longrightarrow \kappa}\,(ke5)}{\kappa{:}\{x?\,\text{-}\,,z\}\langle y\rangle \Longrightarrow \kappa\langle y\rangle}\,(db1)$$

where $\kappa\langle y\rangle = \kappa\langle y\rangle$. In the third case:

$$\frac{\dfrac{\kappa \stackrel{!}{\Longrightarrow} \kappa' \quad x \stackrel{k}{\longrightarrow} x' \quad k \neq 0}{\kappa{:}\{x?y,\,\text{-}\,\} \stackrel{*}{\Longrightarrow} \kappa\langle y\rangle}\,(br3)}{\kappa{:}\{x?y,\,\text{-}\,\}\langle z\rangle \Longrightarrow \kappa\langle y\rangle}\,(db9)$$

where $\kappa\langle\!\lbrace y\rbrace\!\rangle = \kappa\langle\!\lbrace y\rbrace\!\rangle$.

Case 2f: (if3)

In this case, $d = \kappa\langle\!\lbrace \text{if } x \text{ then } y \text{ else } z\rbrace\!\rangle$ and d' can be any one of $\kappa\colon\{\text{ - }?y, z\}\langle\!\lbrace x\rbrace\!\rangle$, $\kappa\colon\{x?\text{ - }, z\}\langle\!\lbrace y\rbrace\!\rangle$ or $\kappa\colon\{x?y, \text{ - }\}\langle\!\lbrace z\rbrace\!\rangle$. The proof has the form:

$$\dfrac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad \dfrac{x \overset{0}{\longrightarrow} x'}{\text{if } x \text{ then } y \text{ else } z \longrightarrow z}\,(if3)}{\kappa\langle\!\lbrace \text{if } x \text{ then } y \text{ else } z\rbrace\!\rangle \Longrightarrow \kappa\langle\!\lbrace z\rbrace\!\rangle}\,(db2)$$

In the first case:

$$\dfrac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \overset{0}{\longrightarrow} x'}{\kappa\colon\{\text{ - }?y, z\}\langle\!\lbrace x\rbrace\!\rangle \Longrightarrow \kappa\langle\!\lbrace z\rbrace\!\rangle}\,(db8),$$

where $\kappa\langle\!\lbrace z\rbrace\!\rangle = \kappa\langle\!\lbrace z\rbrace\!\rangle$. In the second case:

$$\dfrac{\dfrac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \overset{0}{\longrightarrow} x}{\kappa\colon\{x?\text{ - }, z\} \overset{*}{\Longrightarrow} \kappa\langle\!\lbrace z\rbrace\!\rangle}\,(br2)}{\kappa\colon\{x?\text{ - }, z\}\langle\!\lbrace y\rbrace\!\rangle \Longrightarrow \kappa\langle\!\lbrace z\rbrace\!\rangle}\,(db9),$$

where $\kappa\langle\!\lbrace z\rbrace\!\rangle = \kappa\langle\!\lbrace z\rbrace\!\rangle$.

In the third case:

$$\dfrac{\dfrac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad x \overset{0}{\longrightarrow} x'}{\kappa\colon\{x?y, \text{ - }\} \Longrightarrow \kappa}\,(ke7)}{\kappa\colon\{x?y, \text{ - }\}\langle\!\lbrace z\rbrace\!\rangle \Longrightarrow \kappa\langle\!\lbrace z\rbrace\!\rangle}\,(db1)$$

where $\kappa\langle\!\lbrace z\rbrace\!\rangle = \kappa\langle\!\lbrace z\rbrace\!\rangle$.

Case 3: (db3)

The second to last rule in the proof can be (sub4), (sub5) (sub6) or (sub7).

Case 3a: (sub4) There are two cases depending on whether $\delta = l$ or $\delta = r$. The proof is of the form:

$$\dfrac{\dfrac{\phi_1}{\kappa \overset{[e/a]}{\Longrightarrow} \kappa'} \quad \dfrac{\dfrac{\phi_2}{x \overset{[e/a]}{\longrightarrow} x'} \quad \dfrac{\phi_3}{y \overset{[e/a]}{\longrightarrow} y'}}{x\ y \overset{[e/a]}{\longrightarrow} x'\ y'}\,(sub4)}{\kappa\langle\!\lbrace x\ y\rbrace\!\rangle \Longrightarrow \kappa'\langle\!\lbrace x'\ y'\rbrace\!\rangle}\,(db3)$$

In the first case:

$$\dfrac{\dfrac{\dfrac{\phi_1}{\kappa \overset{[e/a]}{\Longrightarrow} \kappa'} \quad \dfrac{\phi_3}{y \overset{[e/a]}{\longrightarrow} y'}}{\kappa\colon\{\text{ - }\ y\} \overset{[e/a]}{\Longrightarrow} \kappa'\colon\{\text{ - }\ y'\}}\,(sb2) \quad \dfrac{\phi_2}{x \overset{[e/a]}{\longrightarrow} x'}}{\kappa\colon\{\text{ - }\ y\}\langle\!\lbrace x\rbrace\!\rangle \Longrightarrow \kappa'\colon\{\text{ - }\ y'\}\langle\!\lbrace x'\rbrace\!\rangle}\,(db3)$$

where $\kappa'\llparenthesis x'\ y'\rrparenthesis \overset{l}{\Downarrow} \kappa'\colon\{\text{-}\ \ y'\}\llparenthesis x'\rrparenthesis$. In the second case:

$$\dfrac{\dfrac{\dfrac{\phi_1}{\kappa \overset{[e/a]}{\Longrightarrow} \kappa'} \quad \dfrac{\phi_2}{x \overset{[e/a]}{\longrightarrow} x'}}{\kappa\colon\{x\ \ \text{-}\} \overset{[e/a]}{\Longrightarrow} \kappa'\colon\{x'\ \ \text{-}\}}\,(sb3) \qquad \dfrac{\phi_3}{y \overset{[e/a]}{\longrightarrow} y'}}{\kappa\colon\{x\ \ \text{-}\}\llparenthesis y\rrparenthesis \Longrightarrow \kappa'\colon\{x'\ \ \text{-}\}\llparenthesis y'\rrparenthesis}\,(db3)$$

where $\kappa'\llparenthesis x'\ y'\rrparenthesis \overset{r}{\Downarrow} \kappa'\colon\{x'\ \ \text{-}\}\llparenthesis y'\rrparenthesis$.

Case 3b: (sub5) The proof is of the form:

$$\dfrac{\dfrac{\dfrac{\phi_1}{\kappa \overset{[e/b]}{\Longrightarrow} \kappa'} \quad \dfrac{\phi_2}{y \overset{[e/b]}{\longrightarrow} y'}}{\kappa\colon\{\text{-}\ \ y\} \overset{[e/b]}{\Longrightarrow} \kappa'\colon\colon\{\text{-}\ \ y'\}}\,(sb2) \quad \dfrac{a \overset{a}{\longrightarrow} a \quad \dfrac{\vdots}{x \overset{[a'/a]}{\longrightarrow} x[a'/a]} \quad \dfrac{\vdots}{x[a'/a] \overset{[e/b]}{\longrightarrow} x[a'/a][e/b]} \quad b \neq a}{(\mathbf{fn}\ \ a => x) \overset{[e/b]}{\longrightarrow} (\mathbf{fn}\ \ a' => x[a'/a][e/b])}\,(sub5)}{\kappa\colon\{\text{-}\ \ y\}\llparenthesis(\mathbf{fn}\ \ a => x)\rrparenthesis \Longrightarrow \kappa'\colon\{\text{-}\ \ y'\}\llparenthesis(\mathbf{fn}\ \ a' => x[a'/a][e/b])\rrparenthesis}\,(db3) \quad ,$$

Therefore

$$\dfrac{\dfrac{\dfrac{\phi_1}{\kappa \overset{[e/b]}{\Longrightarrow} \kappa'} \quad \dfrac{\phi_2}{y \overset{[e/b]}{\longrightarrow} y'} \quad b \neq a}{\kappa\colon\{y/a'\} \overset{[e/b]}{\Longrightarrow} \kappa'\colon\{y'/a'\}}\,(sb7) \qquad \dfrac{\vdots}{x[a''/a] \overset{[e/b]}{\longrightarrow} x[a''/a][e/b]}}{\kappa\colon\{y/a''\}\llparenthesis x[a''/a]\rrparenthesis \Longrightarrow \kappa'\colon\{y'/a''\}\llparenthesis x[a''/a][e/b]\rrparenthesis}\,(db3).$$

Therefore we have

$$\kappa'\colon\{\text{-}\ \ y'\}\llparenthesis(\mathbf{fn}\ \ a' => x[a'/a][e/b])\rrparenthesis \overset{\lambda}{\Downarrow} \kappa'\colon\{y'/a''\}\llparenthesis x[a'/a][e/b][a''/a']\rrparenthesis.$$

and since

$$x[a'/a][e/b][a''/a']\ S\ x[a''/a][e/b],$$

we have that

$$\kappa'\colon\{y'/a''\}\llparenthesis(\mathbf{fn}\ \ a' => x[a''/a])\rrparenthesis\ R'\ \kappa'\colon\{y'/a''\}\llparenthesis(\mathbf{fn}\ \ a' => x[a'/a][e/b][a''/a'])\rrparenthesis.$$

Case 3c: (sub6)
The proof is of the form:

$$\dfrac{\dfrac{\dfrac{\phi_1}{\kappa \overset{[e/a]}{\Longrightarrow} \kappa'} \quad \dfrac{\phi_2}{y \overset{[e/a]}{\longrightarrow} y'}}{\kappa\colon\{\text{-}\ \ y\} \overset{[e/a]}{\Longrightarrow} \kappa'\colon\{\text{-}\ \ y'\}}\,(sb2) \qquad \dfrac{}{(\mathbf{fn}\ \ a => x) \overset{[e/a]}{\longrightarrow} (\mathbf{fn}\ \ a => x)}\,(sub6)}{\kappa\colon\{\text{-}\ \ y\}\llparenthesis(\mathbf{fn}\ \ a => x)\rrparenthesis \Longrightarrow \kappa'\colon\{\text{-}\ \ y'\}\llparenthesis(\mathbf{fn}\ \ a => x)\rrparenthesis}\,(db3)$$

Therefore:

$$\dfrac{\dfrac{\dfrac{\phi_1}{\kappa \overset{[e/a]}{\Longrightarrow} \kappa'} \qquad \dfrac{\phi_2}{y \xrightarrow{[e/a]} y'}}{\kappa\colon \{y/a'\} \Longrightarrow \kappa'\colon \{y'/a'\}}\,(ke8)}{\kappa\colon \{y/a'\}\langle\!|x[a'/a]|\!\rangle \Longrightarrow \kappa'\colon \{y'/a'\}\langle\!|x[a'/a]|\!\rangle}\,(db1)$$

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

Case 3d: (sub7) There are three cases depending on whether $\delta =$ if, then or else. The proof is of the form:

$$\dfrac{\kappa \overset{[e/a]}{\Longrightarrow} \kappa' \quad \dfrac{x \xrightarrow{[e/a]} x' \quad y \xrightarrow{[e/a]} y' \quad z \xrightarrow{[e/a]} z'}{\text{if } x \text{ then } y \text{ else } z \xrightarrow{[e/a]} \text{if } x' \text{ then } y' \text{ else } z'}\,(sub7)}{\kappa\langle\!|\text{if } x \text{ then } y \text{ else } z|\!\rangle \Longrightarrow \kappa'\langle\!|\text{if } x' \text{ then } y' \text{ else } z'|\!\rangle}\,(db3) \quad .$$

In the first case:

$$\dfrac{\dfrac{\kappa \overset{[e/a]}{\Longrightarrow} \kappa' \quad y \xrightarrow{[e/a]} y' \quad z \xrightarrow{[e/a]} z'}{\kappa\colon \{\text{ - }?y, z\} \overset{[e/a]}{\Longrightarrow} \kappa'\colon \{\text{ - }?y', z'\}}\,(sb4) \qquad x \xrightarrow{[e/a]} x'}{\kappa\colon \{\text{ - }?y, z\}\langle\!|x|\!\rangle \Longrightarrow \kappa'\colon \{\text{ - }?y', z'\}\langle\!|x'|\!\rangle}\,(db3)$$

where $\kappa\langle\!|\text{if } x' \text{ then } y' \text{ else } z'|\!\rangle \overset{\text{if}}{\Downarrow} \kappa\colon \{\text{ - }?y', z'\}\langle\!|x'|\!\rangle$. In the second case:

$$\dfrac{\dfrac{\kappa \overset{:}{\Longrightarrow} [e/a]\kappa' \quad x \xrightarrow{[e/a]} x' \quad z \xrightarrow{[e/a]} z'}{\kappa\colon \{x?\text{ - }, z\} \overset{[e/a]}{\Longrightarrow} \kappa'\colon \{x'?\text{ - }, z'\}}\,(sb5) \qquad y \xrightarrow{[e/a]} y'}{\kappa\colon \{x?\text{ - }, z\}\langle\!|y|\!\rangle \Longrightarrow \kappa'\colon \{x'?\text{ - }, z'\}\langle\!|y'|\!\rangle}\,(db3)$$

where $\kappa\langle\!|\text{if } x' \text{ then } y' \text{ else } z'|\!\rangle \overset{\text{then}}{\Downarrow} \kappa\colon \{x'?\text{ - }, z'\}\langle\!|y'|\!\rangle$. In the third case:

$$\dfrac{\dfrac{\kappa \overset{[e/a]}{\Longrightarrow} \kappa' \quad x \xrightarrow{[e/a]} x' \quad y \xrightarrow{[e/a]} y'}{\kappa\colon \{x?y, \text{ - }\} \overset{[e/a]}{\Longrightarrow} \kappa'\colon \{x'?y', \text{ - }\}}\,(sb6) \qquad z \xrightarrow{[e/a]} z'}{\kappa\colon \{x?y, \text{ - }\}\langle\!|z|\!\rangle \Longrightarrow \kappa'\colon \{x'?y', \text{ - }\}\langle\!|z'|\!\rangle}\,(db3)$$

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

Case 4: (db4)

The proof is of the form:

$$\dfrac{\kappa \overset{!}{\Longrightarrow} \kappa' \quad \dfrac{x \xrightarrow{[v/a]} x[v/a]}{(\mathbf{fn}\ a =\!> x) \xrightarrow{\Box v} x[v/a]}\,(tp5)}{\kappa\colon \{\text{ - }\ v\}\langle\!|(\mathbf{fn}\ a =\!> x)|\!\rangle \Longrightarrow \kappa\langle\!|x[v/a]|\!\rangle}\,(db4)$$

Therefore

$$\cfrac{\cfrac{\kappa \overset{!}{\Longrightarrow} \kappa}{\kappa\colon \{v/a'\} \overset{[v/a']}{\Longrightarrow} \kappa}\,(sb1) \qquad x[a'/a] \xrightarrow{[v/a']} x[a'/a][v/a']}{\kappa\colon \{v/a'\}\langle\!\langle x[a'/a]\rangle\!\rangle \Longrightarrow \kappa\langle\!\langle x[a'/a][v/a']\rangle\!\rangle}\,(db3)$$

where $\kappa\langle\!\langle x[v/a]\rangle\!\rangle \; S \; \kappa\langle\!\langle x[a'/a][v/a']\rangle\!\rangle$.

Case 5: (db5),(db6),(db7),(db8)

In all of these cases, the expression in focus must be a constant or function definition and no focusing step is possible.

Case 6: (db9) First we will consider if $\delta \in \{\ l, r,$ if, then or else $\}$ then we will consider the case where $\delta = \lambda$.

Case 6a: $d = \kappa\langle\!\langle x\rangle\!\rangle$, $d' = \kappa\colon c\langle\!\langle x'\rangle\!\rangle$, $\delta \in \{\ l, r,$ if, then or else $\}$

$$\cfrac{\cfrac{\phi}{\kappa \overset{*}{\Longrightarrow} d}}{\kappa\langle\!\langle x\rangle\!\rangle \Longrightarrow d}\,(db9)$$

Therefore

$$\cfrac{\cfrac{\cfrac{\phi}{\kappa \overset{*}{\Longrightarrow} d}}{\kappa\colon c \overset{*}{\Longrightarrow} d}\,(br1)}{\kappa\colon c\langle\!\langle x'\rangle\!\rangle \Longrightarrow d}\,(db9)$$

where $d = d$.

Case 6b: $d = \kappa\colon \{\ \text{-}\ \ y\}\langle\!\langle(\mathbf{fn}\ a \mathrel{=>} x)\rangle\!\rangle$, $d' = \kappa\colon \{y/a'\}\langle\!\langle x[a'/a]\rangle\!\rangle$, $\delta = \lambda$

$$\cfrac{\cfrac{\cfrac{\phi}{\kappa \overset{*}{\Longrightarrow} d}}{\kappa\colon \{\ \text{-}\ \ y\} \overset{*}{\Longrightarrow} d}\,(br1)}{\kappa\colon \{\ \text{-}\ \ y\}\langle\!\langle x\rangle\!\rangle \Longrightarrow d}\,(db9)$$

Therefore

$$\cfrac{\cfrac{\cfrac{\phi}{\kappa \overset{*}{\Longrightarrow} d}}{\kappa\colon \{y/a\} \overset{*}{\Longrightarrow} d}\,(br1)}{\kappa\colon \{y/a'\}\langle\!\langle x[a'/a]\rangle\!\rangle \Longrightarrow d}\,(db9)$$

where $d = d$.

Now we will consider the possible transitions of d'. Actually, we will see that we have already considered all the necessary cases.

Case 1:(db1) This case was considered as Case 1 in the first part of this proof.
Case 2:(db2) The last rule in the proof can be
(tr1): considered in Case 2a
(tr2): considered in Case 2b
(tr3): considered in Case 2d
Case 3:(db3) The last rule in the proof can be (sb1) - (sb7)
(sb1): considered in Case 4
(sb2): considered in Case 3a
(sb3): considered in Case 3a
(sb4): considered in Case 3d
(sb5): considered in Case 3d
(sb6): considered in Case 3d
(sb7): considered in Case 3b
Case 4:(db4) This case was considered in Case 2c.
Case 5:(db5) This case was considered in Case 2a.
Case 6:(db6) This case was considered in Case 2c.
Case 7:(db7) This case was considered in Case 2e.
Case 8:(db8) This case was considered in Case 2f.
Case 9:(db9) The last rule in the proof can be (br1) - (br3)
(br1): considered in Case 6.
(br2): considered in Case 2f.
(br3): considered in Case 2e.

□